MILITARY STANDARD

Instruction Set Architecture

for the Military Computer Family

FSC IPSC

Table of Contents

1. SCOPE AND PURPOSE 1
1.1. Scope 1
1.2. Purpose 1

2. APPLICABLE DOCUMENTS 2

3. DEFINITIONS 3

4. OVERVIEW 4
4.1. Memory Organization 4
4.2. Instruction Format 4
4.3. Control Structure 4
4.4. Data Types 4

5. OPERAND ADDRESSING MODES 7
5.1. Maximum Accessible Register 7
5.2. Recursive Modes 7
5.3. Address Operands 7
5.4. Register Mode 8
5.5. Short Literal Mode 9
5.6. Literal Mode 9
5.7. Memory Addressing Modes 10
5.7.1. Indirect Register Mode 11
5.7.2. Register Indexed Modes 12
5.7.3. Absolute Mode 13
5.8. Parameter Addressing Modes 14
5.8.1. Short Parameter Mode 14
5.8.2. Parameter Mode 14
5.9. Unscaled Index Mode 15
5.10. Scaled Index Mode 16

6. PROCESSOR STATUS WORD 18
6.1. Kernel/Task Mode 18
6.2. Last Mode 18
6.3. Reserved Bits 18
6.4. Priority 18
6.5. Carry Condition Code 18
6.6. Truncate Condition Code 18
6.7. Negative Condition Code 19
6.8. Zero Condition Code 19
6.9. Debugging Control 19
6.10. Privilege 19
6.11. Base of Context 19
6.12. Supervisor Mode 19
6.13. Up/Down Level Exception Propagation 19
6.14. Enable Arithmetic Error 19

6.15. Maxreg Field 19
6.16. Number of Parameters 19

7. AUXILIARY STATUS WORD 20
7.1. Soft Memory Error Enable 20
7.2. Reserved Bits 20
7.3. Floating Point Mode Control 20
7.3.1. Infinity Control 20
7.3.2. Exception Event and Mask Bits 20

8. PROCEDURE INTERFACE 21
8.1. The Context Stacks 21
8.1.1. Context Pointers 21
8.1.2. Structure of Context Stacks 22
8.1.3. Cacheing of the Context Stack 23
8.1.4. Changing of Context Stack Pointers 23
8.1.5. Alignment of Context Pointers 23
8.2. Procedure Descriptor 23
8.2.1. Fixed/Variable Number of Parameters 23
8.2.2. Exception Handler 23
8.2.3. Exception Propagation Control 23
8.2.4. Arithmetic Error Control 24
8.2.5. Available Registers 24
8.3. Procedure Invocation 24
8.3.1. Determination of a New PSW 24
8.3.2. Register Set Allocation 25
8.3.3. Definition of Exception Handler 25
8.3.4. Initialization of Parameter List 25
8.4. Parameter Lists 25
8.4.1. Parameter Specification 26
8.4.2. Parameter Access 26
8.4.3. Example of Parameter Linkage 26
8.4.4. Mechanism of Parameter Addressing 27
8.5. Vectored Calls: SVC and OPEX 30
8.6. Vector Format 30

9. EXCEPTIONS 32
9.1. Procedure Exception Handler 32
9.2. Definition of an Exception Handler 32
9.3. Raising an Exception 33
9.4. Exception Handling 33
9.5. Supervisor Exception Handler 33

10. DEBUGGING FACILITIES 36
10.1. Program Tracing 36

11. INTERRUPTS AND TRAPS 37
11.1. Interrupt Priority 37
11.2. I/O Interrupts 37
11.3. Software Interrupt Requests 37
11.4. Power Failure Interrupt 37
11.5. Power Restore Interrupt 38

11.6. Memory System Error Traps 39
11.6.1. Hard Memory Errors 39
11.6.2. Soft Memory Errors 39
11.7. Privileged Instruction Trap 39
11.8. Memory Management Traps 39

12. MEMORY MANAGEMENT SYSTEM 40
12.1. Virtual Address Space 40
12.2. Mapping of Virtual Addresses 41
12.2.1. Map Pointer Registers 41
12.2.2. Memory Map Structure 41
12.2.3. Segment Association 41
12.2.4. Relocation of Virtual Addresses 42
12.2.5. Access Protection 42
12.2.6. Protection of the Supervisor 44
12.3. Cacheing of Memory Maps 44
12.4. Memory Management Traps 44
12.5. I/O Space Selection 45
12.6. Subsetting of Memory Management 45

13. I/O CONTROLLERS 46
13.1. Channel Configuration Registers 46
13.2. IOC Programs 46
13.3. Program Counter 46
13.4. Message Pointer Register 46
13.5. Instruction Execution 46
13.6. Operand Accessing 46
13.7. Channel Program Sequencing 47
13.8. IOC Interrupts 47
13.8.1. IOC Error Interrupts 47
13.9. IOC Instructions 47
13.9.1. Transfer Instructions 48
13.9.1.1. Indirect Buffer Specification 48
13.9.2. Channel Status Instructions 53
13.9.3. Control Instructions 55

14. TIMER SUPPORT 59
14.1. Time of Day 59
14.2. Interval Timers 59

15. ASSIGNED PHYSICAL ADDRESSES 60
15.1. Interrupt Vector Assignments 60
15.2. I/O Space Assignments 60

16. INTEGER ARITHMETIC 61
16.1. Integer Data Types 61
16.2. Integer Arithmetic Instructions 62

17. LOGICAL INSTRUCTIONS 76

18. SHIFT AND ROTATE INSTRUCTIONS 79

19. MOVE AND CLEAR INSTRUCTIONS 82

20. COMPARE AND TEST INSTRUCTIONS 87

21. CONTROL INSTRUCTIONS 92

22. PROCEDURE CALL AND RETURN INSTRUCTIONS 103

23. TASK CONTROL INSTRUCTIONS 108

24. EXCEPTION HANDLING INSTRUCTIONS 113

25. LIST INSTRUCTIONS 117

26. STRING INSTRUCTIONS 120

27. BIT FIELD INSTRUCTIONS 124

28. MISCELLANEOUS INSTRUCTIONS 130

29. FLOATING-POINT ARITHMETIC 135
29.1. Floating-point Data Classes 135
29.2. Floating-point Formats 135
29.3. Floating-Point Exceptions 136
29.3.1. Illegal.Operation 137
29.3.2. Divide.By.Zero 137
29.3.3. Floating.Underflow 137
29.3.4. Floating.Overflow 138
29.3.5. Floating.Inexact 138
29.4. Rounding 138
29.5. Infinity Arithmetic 138
29.6. Floating-Point Instructions 139

30. UNDEFINED OPCODES 158

I. ISP Description 159

List of Figures

Figure 4-1:	Instruction Format	4
Figure 4-2:	Integer Data Types	4
Figure 4-3:	Floating Point Data Types	5
Figure 5-1:	Operand Specifiers	7
Figure 6-1:	Processor Status Word	18
Figure 7-1:	Auxiliary Status Register	20
Figure 8-1:	Stacked Procedure Contexts	21
Figure 8-2:	Procedure Descriptor	23
Figure 8-3:	Example of a Call Context	26
Figure 8-4:	Normal Operand Processing	27
Figure 8-5:	Parameter List Operand Processing	27
Figure 8-6:	Parameter Access	27
Figure 8-7:	SVC and OPEX Vector Registers	30
Figure 8-8:	Vectored Call Address Calculation	30
Figure 8-9:	Vector Format	30
Figure 9-1:	States of an Exception Handler	32
Figure 9-2:	Transfer to an Exception Handler	33
Figure 11-1:	Vectoring of I/O Interrupts	37
Figure 12-1:	Map Data Structure	40
Figure 12-2:	Memory Map Pointer Registers	41
Figure 12-3:	Map Entry Format	41
Figure 12-4:	Address Relocation	42
Figure 13-1:	Indirect Buffer List	48
Figure 13-2:	Channel Program Status	53
Figure 14-1:	Timer Control Register	59
Figure 29-1:	Floating Point Number Representation	135
Figure 29-2:	Exception Bits in ASR	136
Figure 29-3:	The Compare Operation	151

1. Scope and Purpose

1.1. Scope. This standard defines the instruction set architecture for the Military Computer Family (MCF) of the U.S. Army. The instruction set architecture includes all information required by a programmer in order to write any time independent program which will execute on computers conforming to this standard.

1.2. Purpose. The purpose of this document is to define the Military Computer Family Architecture, independent of any specific implementation or vendor, with sufficient precision to permit independent implementations of this architecture that execute identical programs in the identical manner.

2. Applicable Documents

No other documents are specified herein.

3. Definitions

The following terms are used in this document:

Implementation Dependent
: The behavior of an action or feature described as implementation dependent is not defined by this document. The actual behavior seen by the programmer is dependent on the hardware he is using. This behavior will differ between machines.

Cacheing
: The term cacheing is used to refer to a variety of implementation techniques for enhancing performance by maintaining the active copy of some quantity in a place other than its assigned storage. The technique invoked may be as simple as loading and storing of a register (file) or as complex as an associative memory scheme.

Reserved (value)
: A reserved value is a bit pattern which, when written into a particular bit field, will cause unpredictable results. The use of such values by the software is forbidden.

Reserved (bit field)
: A reserved bit field is a sequence of bits which may produce a **Specification.Error** exception or unpredictable actions if set to a value other than zero. Reserved bits or reserved bit fields yield undefined results if read.

Undefined
: An undefined result may produce arbitrary bit patterns in all fields that it was specified as modifying.

Unpredictable
: An unpredictable action may produce any change in the state of the machine which is consistent with the privileges of the program that caused it. For example, an unpredictable operation performed by a user task may destroy any of the locations he can access normally, but may not destroy any state protected by the privilege mechanisms. This restriction may be breached by a supervisor which gives a user access to critical memory or functions. For this reason, access to the CPU registers in the I/O space is considered a privileged operation.

It should be noted that while the programmer cannot rely on any properties of an unpredictable action, it is considered desirable to make such actions as innocuous as is practical.

4. Overview

The MCF architecture is a 32 bit general register architecture. Some of the characteristics of the architecture are:

- Variable length instructions with variable numbers of operands.
- Explicit addressing of all instruction operands.
- Multiple addressing modes for register, literal, memory and parameter access.
- Procedure based control structure with a local register set for each procedure.
- Support for several data types and representation lengths.
- Memory mapped I/O control.
- Vectored Interrupts and Exceptions.

4.1. Memory Organization. Main memory is byte addressable. The architecture provides for a virtual address space of 2^{32} bytes. Bits are numbered from left to right; Memory words are addressed by the address of their most significant byte. Multi-byte data (halfwords, words, doublewords) may be aligned on arbitrary byte boundaries, although performance improvements may be expected with aligned data. The upper 2^{20} bytes of the physical address space are called I/O space. These addresses are used to access device and channel control registers.

4.2. Instruction Format. Instructions are specified by a one byte opcode followed by multi-byte operand specifiers. Each of these operand specifiers allows fully general addressing of the operand. The number of operand specifiers used by an instruction is determined solely by the nature of the operation it performs. No arbitrary restrictions are placed on the number of operands an instruction can address.

Figure 4-1: Instruction Format

4.3. Control Structure. The architecture provides a high level procedure based control structure. Procedures may be invoked by calls, interrupts, traps, or as independent tasks. The procedure mechanism automatically provides for the passing of parameters and the maintenance of control linkages. Each procedure is provided with its own register set of up to 16 registers (one of which is subsumed by the program counter) for local use. The protection provided by the procedure structure allows privileged code to communicate with non-privileged code in a meaningful way.

4.4. Data Types. The architecture provides support for integer, logical, and floating data types with multiple size representations for each. Integers are fully supported in 8, 16, and 32 bit lengths and some extended precision support is provided for 64 bit lengths. Integers are represented in 2's complement notation.

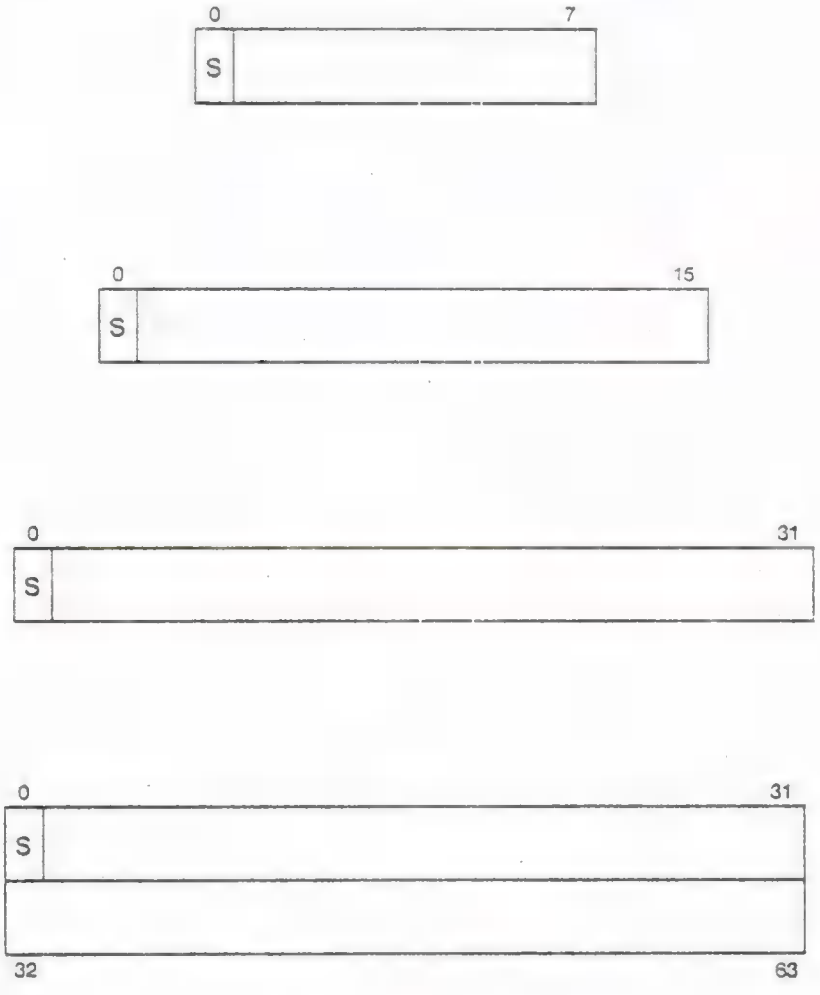


Figure 4-2: Integer Data Types

Logical quantities are supported in 1, 8, 16, and 32 bit lengths. These data types may be used interchangeably as unsigned integers and bit patterns. The ability to access signed and unsigned bit fields is also provided.

Floating point numbers are provided in single (32-bit) and double (64-bit) length representations. The representations adopted are those of the IEEE floating point standard. Conversion between integer and floating point representations is provided. Conversion between different lengths is provided implicitly; No conversion instructions are necessary.

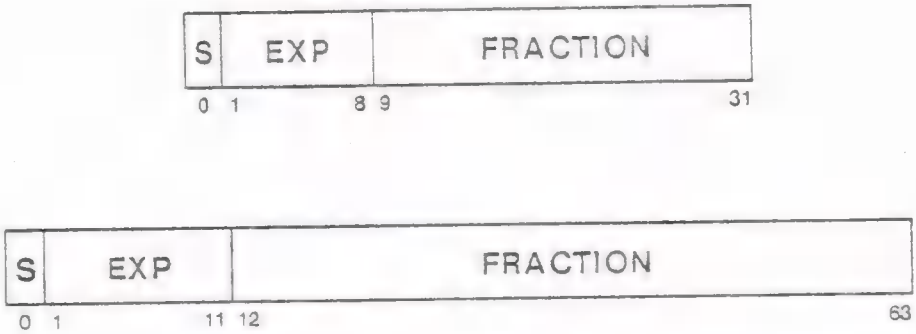


Figure 4-3: Floating Point Data Types

5. Operand Addressing Modes

Operands are identified by the instructions through the decoding of multi-byte strings called **operand specifiers**. Operand specifiers contain the information needed to determine the location and the size of the operands to be accessed.

The location of each operand is fully general and may be:

- a literal constant in the code stream
- a register
- a memory location

The operand specifier selects the size of the operand from the supported primitive data types:

- 8 bit integer/logical
- 16 bit integer/logical
- 32 bit integer/logical
- 32 bit floating point
- 64 bit floating point

The overall structure of the MCF architecture places a design restriction on the operand specifiers. The location (address) of all operands must be computable in the absence of any context information (such as single or double) provided by the opcode. This permits operands to be "pre-evaluated" in the absence of such information as required by the procedure interface. Addressing modes are also required to be free of side effects. This eliminates any order dependencies in operand evaluation.

An overview of the addressing modes is shown in figure 5.

5.1. Maximum Accessible Register. Several of the addressing modes specify a four bit register field. This field is checked against the *Maxreg* field of the PSW to insure that the specified register is available in the current context. If the specified register number is greater than Maxreg, an **Illegal.register** exception shall be initiated.

5.2. Recursive Modes. Three addressing modes, Parameter, Unscaled Index, and Scaled Index, use other operand specifiers as a "subpart" of their complete operand specification. These are refered to as **recursive** modes. The subpart operand specifiers (which form a part of the full recursive mode specifier) shall be restricted to non-recursive modes. For example, as part of its specification the unscaled index mode calls for two other operands to be specified and used as part of the calculation of the unscaled indexed operand. These two subcart operands cannot be unscaled index or either of the other two recursive operands. If a recursive type operand is encountered when a subpart operand is being decoded, an **Illegal.mode** exception shall be initiated.

5.3. Address Operands. Addressing modes that have their operands in memory may be used as **address operands**. When an instruction calls for an address operand, the operand specifier is evaluated until the address of the operand is obtained. That *address* is returned to the instruction rather than value of the operand. If an operand specifier whose operand is NOT in memory (e.g. register mode) is specified as an address operand, an **Illegal.address** exception shall be initiated.

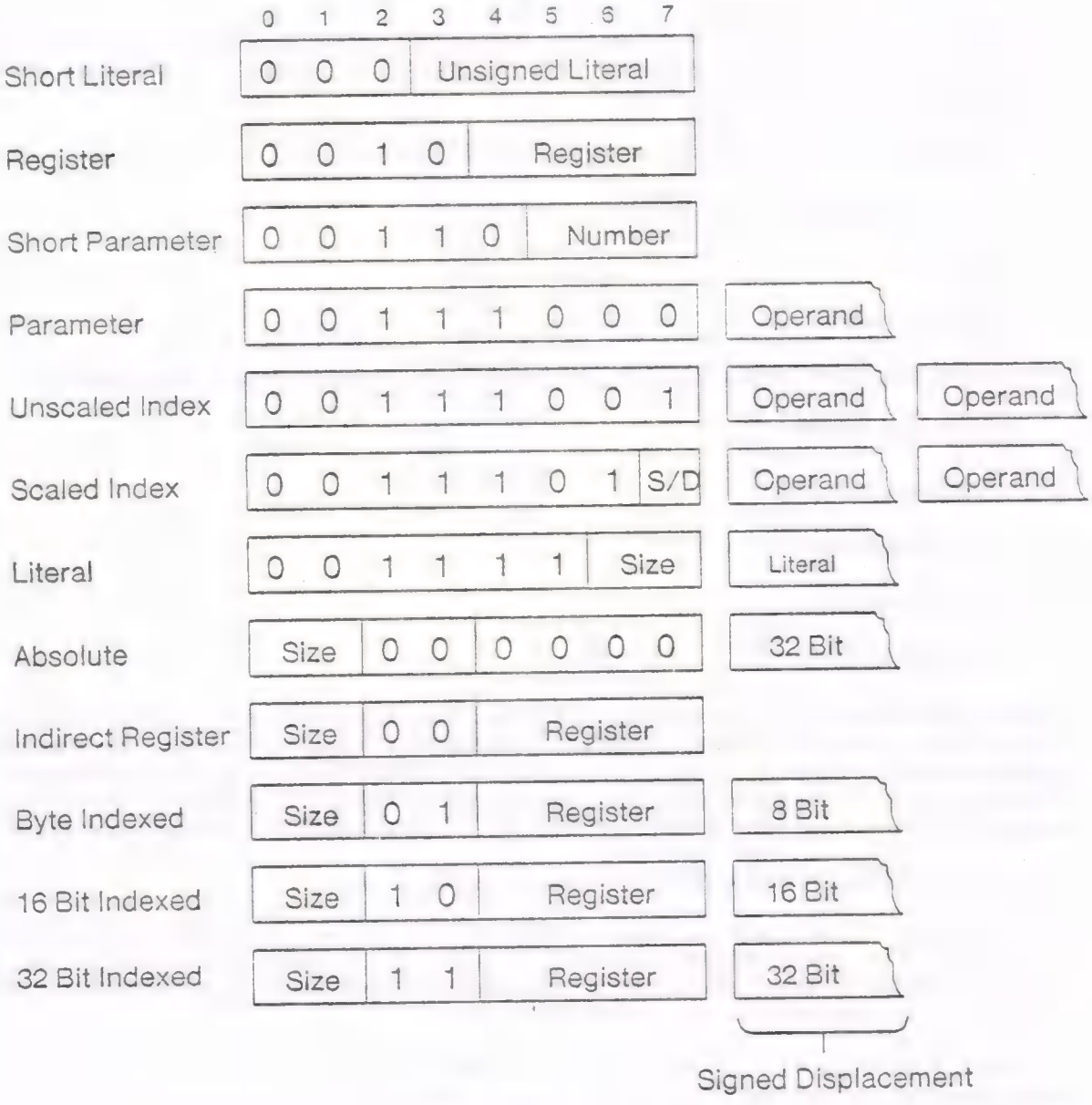
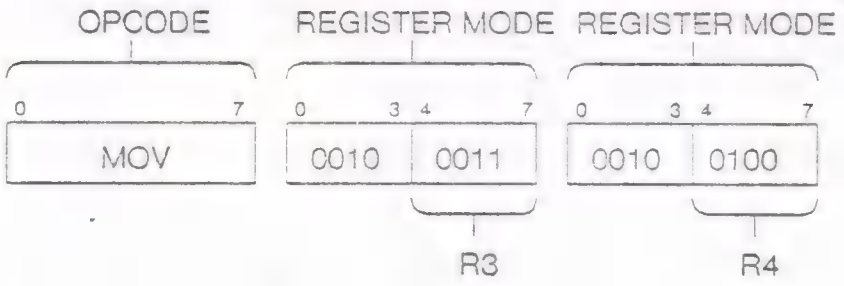

0 1 2 3 4 5 6 7
Short Literal
0 0 0 Unsigned Literal
Register
0 0 1 0 Register
Short Parameter
0 0 1 1 0 Number
Parameter
0 0 1 1 1 0 0 0 Operand
Unscaled Index
0 0 1 1 1 0 0 1 Operand Operand
Scaled Index
0 0 1 1 1 0 1 S/D Operand Operand
Literal
0 0 1 1 1 1 Size Literal
Absolute
Size 0 0 0 0 0 0 32 Bit
Indirect Register
Size 0 0 Register
Byte Indexed
Size 0 1 Register 8 Bit
16 Bit Indexed
Size 1 0 Register 16 Bit
32 Bit Indexed
Size 1 1 Register 32 Bit
Signed Displacement

Figure 5-1: Operand Specifiers

5.4. Register Mode. In register mode the operand is located in the register specified by the second half (bits 4:7) of the operand specifier byte. Register operands shall be word (32 bit) size and therefore require no size field specification. The Maxreg restriction applies to register mode. Access to register 0 (the program counter) using register mode is prohibited and shall cause an **illegal.register** exception. The operand specifier associated with register mode 0 (00100000) is reserved for future extensions.

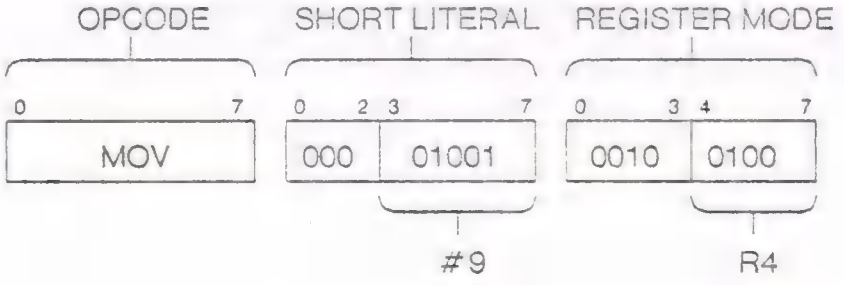
EXAMPLE: MOV R3, R4



In this example, the contents of register 3 are copied into register 4.

5.5. **Short Literal Mode.** In order to facilitate the use of short literals, a single byte specifier allowing unsigned literals of up to 5 bits is provided. Bits 3:7 of the specifier byte form the short literal. This short literal (0 - 31 decimal) is zero extended internally to the size necessary for further computation. An attempt to write into a short literal mode operand shall generate an **Illegal.write** exception.

EXAMPLE: MOV #9, R4



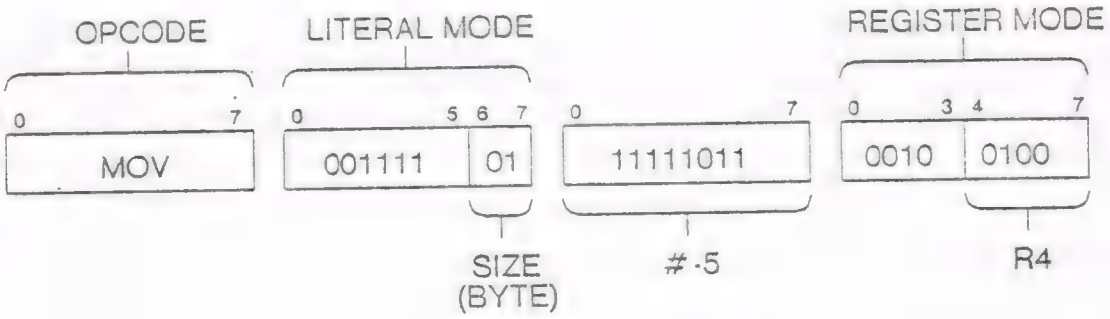
The literal 9 (encoded in the lower part of the short literal specifier) is zero extended and placed in register 4.

5.6. **Literal Mode.** When a literal that is negative or too large for short literal mode (larger than 31) must be specified. Literal Mode may be used. Literal mode addressing allows the specification of literals that are one, two, four, or eight bytes long. The literal follows the first specifier byte in the code stream. Bits 6:7 of the first specifier byte contain the encoded size of the literal. The encoding is:

- 00 - Double word (8 bytes, 64 bits)
- 01 - Byte (8 bits)
- 10 - Halfword (2 bytes, 16 bits)
- 11 - Word (4 bytes, 32 bits)

No context information (such as Single/Double) is required to specify a literal. Once the literal is fetched, it may be extended internally, based on the requirements of the operation. An attempt to write into a literal mode operand shall generate an Illegal.write exception.

EXAMPLE: MOV #-5, R4



The single byte literal -5 is sign extended and copied into register 4.

5.7. Memory Addressing Modes. The addressing modes described in this section all address memory. They may be used as address operands. The size field of memory operand descriptors is tightly encoded. Operands are classified as single or double length as determined by the opcode. Memory operands may be 1, 2, or 4 bytes for single length operands and 2, 4, or 8 bytes for double length operands as specified by bits 0:1 of the first specifier byte.

Size field encoding for single length:

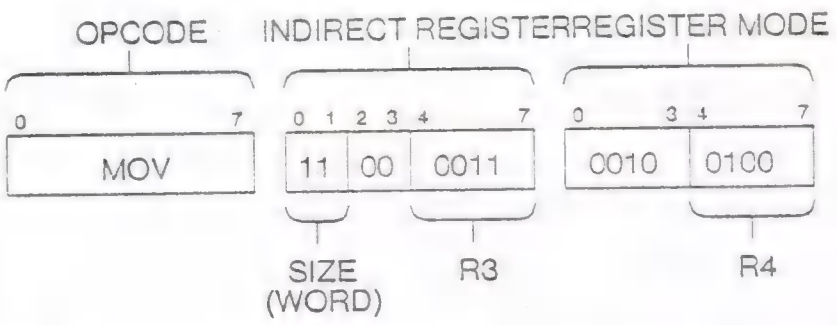
- 00 - Illegal (used to designate other addressing modes)
- 01 - Byte (8 bits)
- 10 - Halfword (2 bytes, 16 bits)
- 11 - Word (4 bytes, 32 bits)

Size field encoding for double length:

- 00 - Illegal (used to designate alternate addressing modes)
- 01 - Halfword (2 bytes, 16 bits)
- 10 - Word (4 bytes, 32 bits)
- 11 - Doubleword (8 bytes, 64 bits)

5.7.1. Indirect Register Mode. In indirect register mode, the second half of the operand specifier byte (bits 4:7) designates a register that contains the address of the operand. The Maxreg restriction applies to the register designation. Note that specification of register 0 (the program counter) indicates absolute addressing described below.

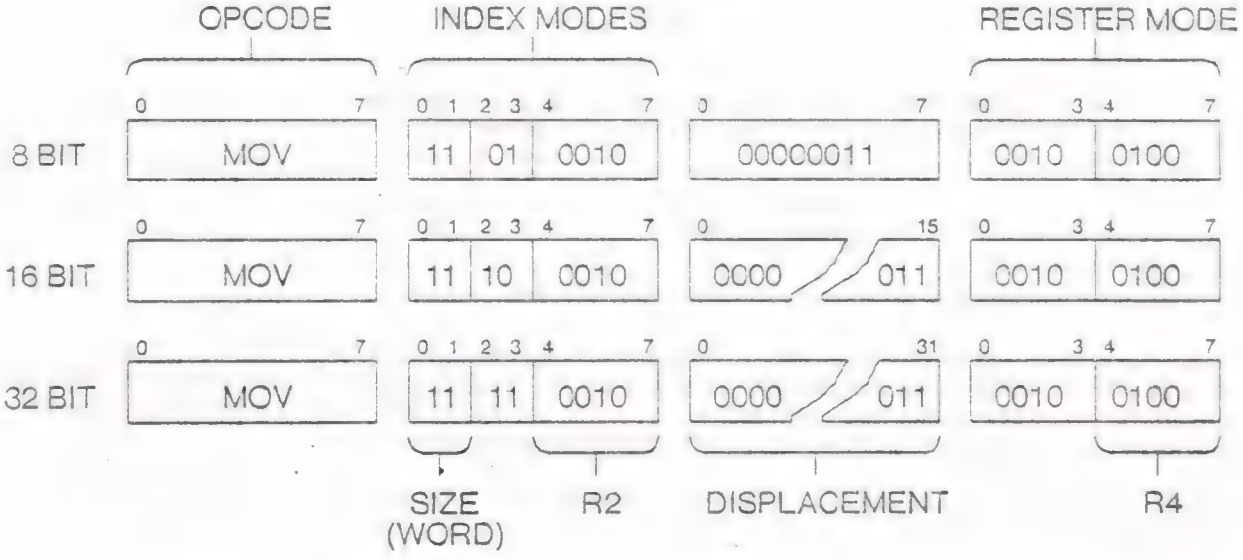
EXAMPLE: MOV @R3↑W, R4



The word operand located at the address specified in register 3 is copied into register 4.

5.7.2. Register Indexed Modes. There are three Register indexed modes, Byte Indexed, Halfword Indexed, and Word Indexed. Each one specifies a register (bits 4:7) and a size (bits 0:1) in the first operand specifier byte followed by a signed displacement that is 8, 16, or 32 bits long respectively. The signed displacement is added to the contents of the designated register to form the address of the operand. The Maxreg restriction applies to the designated register. Register 0 (the program counter) may be designated as the index register in all register index modes.

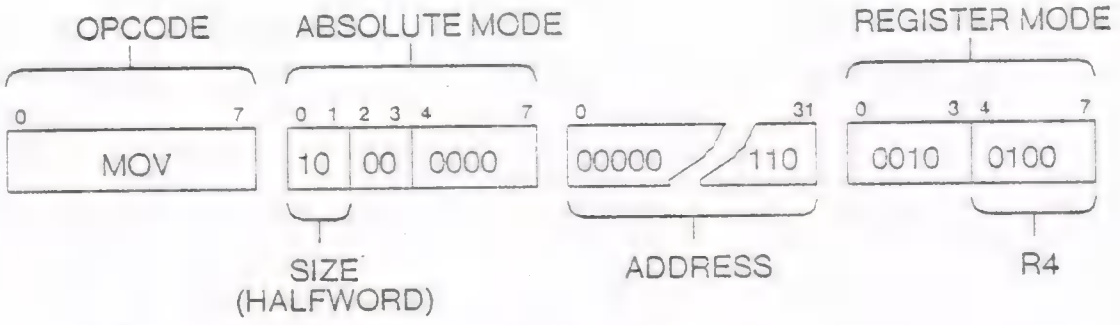
EXAMPLE: MOV 3(R2)W, R4



The displacement 3 is added to the contents of register 2 to form an address. The contents of the word at this address are copied into register 4. Note that this operation is the same for Byte Indexed, Halfword Indexed, and Word Indexed. The only difference is the size of the displacement.

5.7.3. Absolute Mode. In absolute mode, the first specifier byte is followed by the 32 bit address of the operand.

EXAMPLE: MOV @(6)H,R4

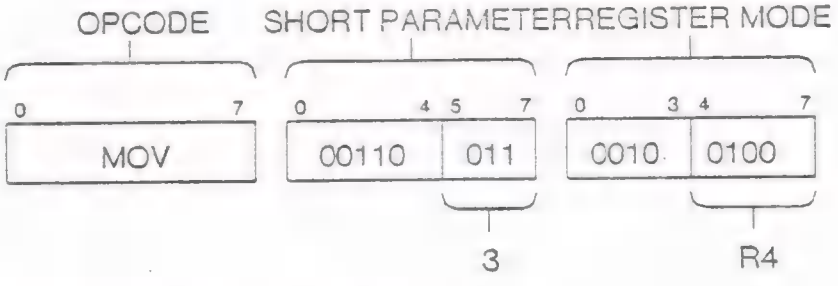


The halfword starting at location 6 in memory is copied into register 4.

5.8. Parameter Addressing Modes. Parameter addressing modes allow access to parameters defined by the caller of a procedure. The parameter modes evaluate to an unsigned integer **parameter number**. Parameter numbers are bound to the actual parameters by the procedure call. See section 8.4 for more information on parameters, their generation and use.

5.8.1. Short Parameter Mode. Bits 5:7 of the specifier byte designate a parameter number in the range 0 to 7.

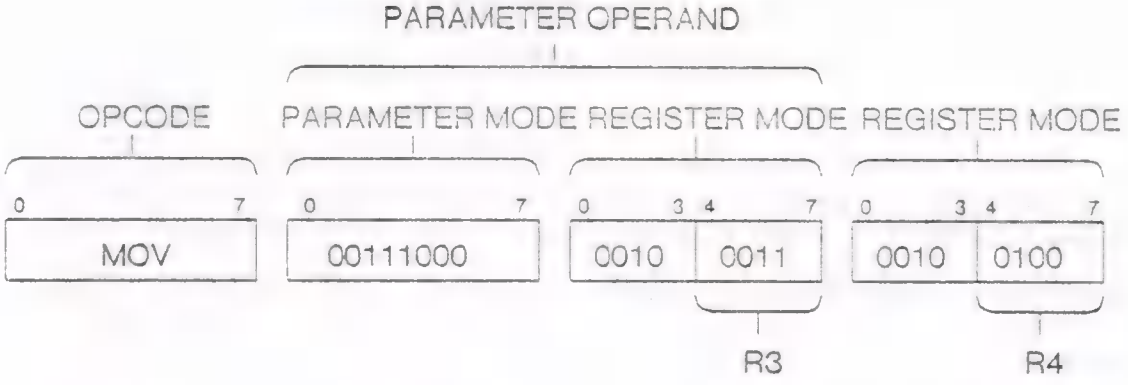
EXAMPLE: MOV ?3, R4



The contents of parameter 3 are copied into register 4.

5.8.2. Parameter Mode. Parameter Mode is one of the *recursive* modes described above. The first byte of the operand specifier is followed by another complete operand specifier. This *second* operand specifier is evaluated as an unsigned integer parameter number.

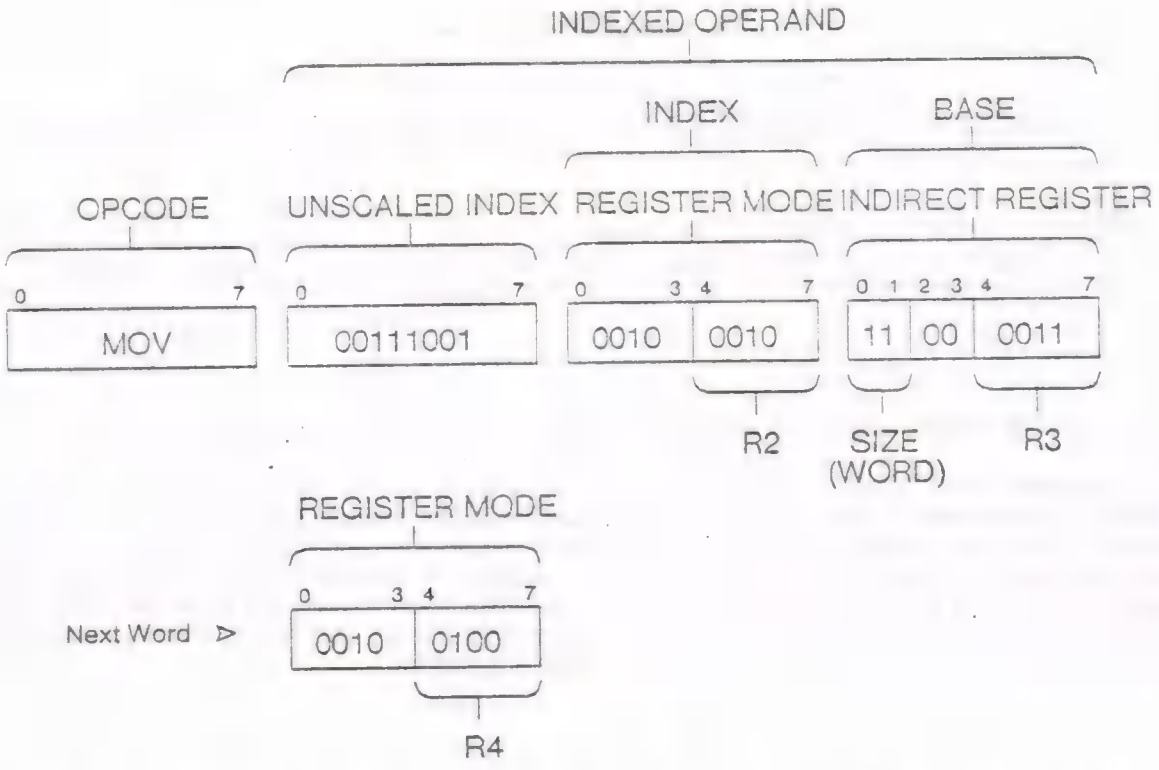
EXAMPLE: MOV ?(R3), R4



The parameter whose number is in register 3 is accessed and its value is copied into register 4.

5.9. Unscaled Index Mode. Unscaled Index is another of the *recursive* modes. The first specifier byte is followed by *two* complete operand specifiers, the first is the **index** and the second is the **base**. The base operand specifier is an *address* type operand, and must evaluate to a memory address as described in section 5.3 above. In order to access the Unscaled Index Operand, the index specifier is evaluated and its **value** is added to the **address** obtained from the base specifier. This forms the full address of the operand. Its size is determined by the size field in the base (second) specifier. Unscaled index mode may be used as an address operand.

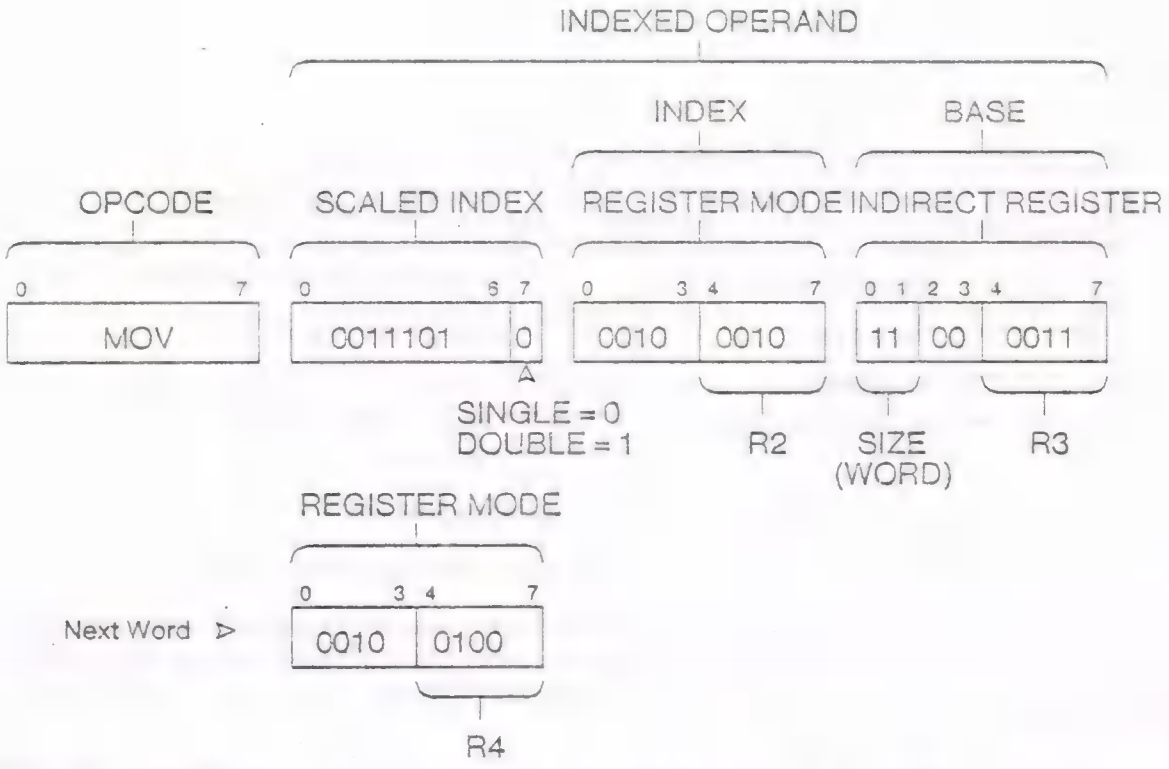
EXAMPLE: MOV @R3(R2)W, R4



The index in register 2 is added to the base address in register 3. The value of the word at this location is copied into register 4.

5.10. Scaled Index Mode. Scaled Index is the third *recursive* mode. It is very similar to unscaled index described above. The first specifier byte is followed by two complete operand specifiers, the index and the base. The base operand specifier is an *address* type operand, and must evaluate to a memory address as described in section 5.3 above. The address of the Scaled index operand is found by evaluating the index specifier, scaling it, and adding it to the address obtained from the base specifier. The scaling is based on the size field in the base specifier and the Single/Double (low order) bit in the first specifier byte. If the Single/Double bit is 0, the scale factor is 1, 2, or 4 based on the base specifier size field. If the Single/Double bit is 1, the scale factor is 2, 4, or 8. Scaling is simply multiplying the index value by the scale factor (since all of the scale factors are powers of 2 the multiplication can be accomplished by shifting). Scaled index mode may be used as an address operand.

EXAMPLE: MOV @R3[R2]W, R4



The index value in register 2 is scaled by 4 (size = word, Single/Double = 0) and added to the base address found in register 3. The value of the word found at the resulting address is copied into register 4.

6. Processor Status Word

Control information related to the current execution context is maintained in the 32 bit Processor Status word. Bits 0:12 contain information global to the currently executing task. Bits 13:31 contain information related to the current procedure's context. The procedure call and return facilities alter only bits 13:31, while initiation or completion of interrupts or tasks alters the entire PSW.

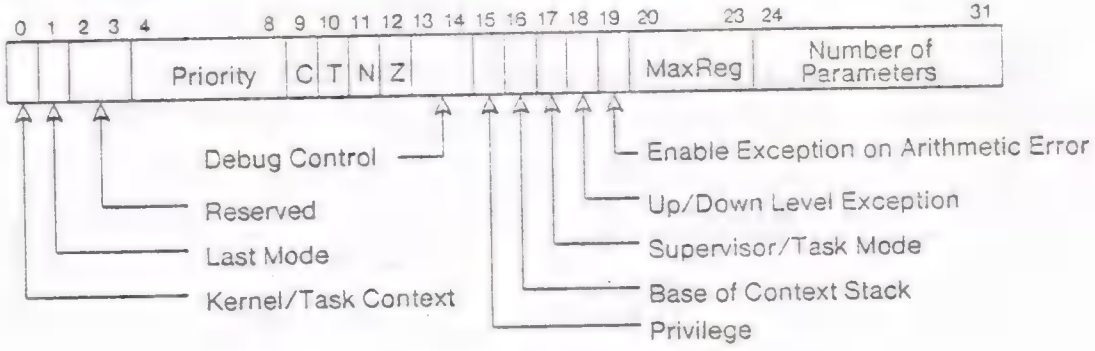


Figure 6-1: Processor Status Word

The form of the PSW is shown in figure 6-1. The fields of the PSW are defined below.

6.1. Kernel/Task Mode. Bit 0 specifies whether the processor is in kernel or task mode. If bit 0 is clear, the processor is in kernel mode. The kernel context stack shall be used for the context of calls. If bit 0 of the PSW is set, the processor is in task mode. The task context stack shall be used for procedure calls.

6.2. Last Mode. Bit 1 shall be set on interrupt and trap operations to reflect the state of the Kernel/Task mode bit prior to the interrupt or trap. This information is used by the RET instruction to determine which context stack is to be reactivated.

6.3. Reserved Bits. Bits 2:3 of the PSW are reserved for implementation dependent functions. A PSW saved as part of an interrupt or trap context change may have an implementation dependent bit pattern in this field. This psw should be restored with this pattern unaltered to restart the interrupted context. The results of altering this bit pattern in such a case are unpredictable.

6.4. Priority. Bits 4:8 of the PSW shall define the current priority of the processor. Hardware or software interrupt requests at an equal or lower priority shall be masked.

6.5. Carry Condition Code. Bit 9 of the PSW is the carry bit. This bit shall be set by integer add and subtract instructions to reflect the carry from the most significant bit of the internal result.

6.6. Truncate Condition Code. Bit 10 of the PSW is the truncation flag. This bit shall be set if the integer result of an operation cannot be correctly represented in the specified result operand. The setting of truncate indicates that significant bits (bits not equal to the stored sign bit) were lost. The truncate bit shall also be set on the occurrence of certain floating point errors. when the corresponding floating point exception is disabled. Refer to section 29.3.

6.7. Negative Condition Code. Bit 11 of the PSW is the negative condition code. This bit shall be set as specified in the instruction descriptions. Generally, setting of the N bit implies a negative or less than condition.

6.8. Zero Condition Code. Bit 12 of the PSW is the zero condition code. This bit shall be set as specified in the instruction descriptions. Generally, setting of the Z bit implies a zero or equal condition.

6.9. Debugging Control. Bits 13:14 of the PSW control the instruction and procedure level debugging facilities. Refer to section 10 for a description of their function.

6.10. Privilege. Bit 15 of the PSW specifies whether the current context is privileged. If this bit is set, privileged instructions may be executed. If this bit is clear, an attempt to execute a privileged instruction shall cause a trap.

6.11. Base of Context. Bit 16 of the PSW indicates the execution context base. If this bit is set, the currently executing context has no caller. Execution of a RET instruction with the Base bit set shall cause an interrupt return operation. Bit 16 shall be set in the PSW of the new context produced by an interrupt, trap, TINIT instruction, or PINIT instruction.

6.12. Supervisor Mode. Bit 17 shall indicate whether the current context is in Supervisor mode. If this bit is set accesses through the supervisor map may be made.

6.13. Up/Down Level Exception Propagation. Bit 18 shall indicate the direction of exception propagation. If bit 18 is clear, the exception shall be directed to the current exception handler or to the caller. If bit 18 is set, the exception shall be directed to the supervisor exception handler. Refer to section 9.4 for a detailed explanation of exception propagation.

6.14. Enable Arithmetic Error. EAE, bit 19 of the PSW, shall control the generation of exceptions related to arithmetic errors. If EAE is set and the T bit (bit 10 of the PSW) is set at the end of an instruction, a **Truncation** exception shall occur. If EAE is clear, truncation exceptions shall be suppressed. EAE is also used as a mask for floating point errors. Refer to section 29.3.

6.15. Maxreg Field. Bits 20:23 of the PSW define the maximum accessible register number in the current context. An attempt to access a register number greater than that specified in Maxreg shall cause an **Illegal.Register** exception.

6.16. Number of Parameters. Bits 24:31 define the number of parameters accessible in the current context. An attempt to access a parameter number greater than that specified in bits 24:31 shall cause an **Illegal.Parameter** exception.

7. Auxiliary Status Word

The auxiliary status word is a 32 bit register located in the I/O space. It contains various processor control and status bits which need be accessed only infrequently. It is not saved and restored on context changes. The format of the ASR is shown in figure 7-1. The fields of the ASR are described below.

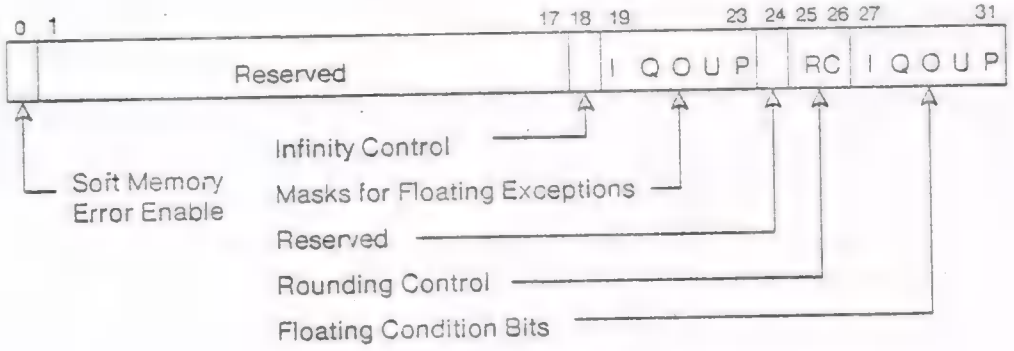


Figure 7-1: Auxiliary Status Register

7.1. Soft Memory Error Enable. Bit 0 shall control the generation of traps on soft memory error conditions. If this bit is clear, no traps shall be generated. Refer to section 11.6 for a description of the memory error handling facility.

7.2. Reserved Bits. Bits 1:17 and 24 are reserved.

7.3. Floating Point Mode Control. Bits 18:23 and 25:31 are dedicated to control and status information for the various modes supported by the IEEE proposed standard floating point. Refer to section 29.3 for a detailed discussion of each.

7.3.1. Infinity Control. Bit 18 shall control the ordering of the special floating point infinity symbols. If set, negative infinity shall be considered less than positive infinity. If clear, negative and positive infinity shall be considered equal.

7.3.2. Exception Event and Mask Bits. For each of five different exception conditions supported by the floating point standard there shall be an event and a mask bit in the ASR. The mask bit shall control generation of an exception on occurrence of the corresponding condition. If the mask bit is set, and EAE is set, an exception shall be generated. If the mask bit is set and EAE is clear, the T bit in the PSW shall be set. If the mask bit is clear the corresponding event bit shall be set and the operation shall be completed. The assigned bits and exceptions for each condition are listed below.

Condition	Mask Bit	Event Bit	Exception
Invalid Operation	19	27	Illegal.Operation
Divide By 0	20	28	Divide.By.Zero
Overflow	21	29	Floating.Overflow
Underflow	22	30	Floating.Underflow
Inexact Result	23	31	Floating.Inexact

8. Procedure Interface

The fundamental unit of execution in the MCF architecture is the procedure. A procedure is a code sequence with an associated context. The context of a procedure includes:

PSW — The processor status word contains information about the current procedure. Included is the number of registers accessible, the number of parameters, the method of dealing with exceptions, and certain privilege information. The context for each procedure includes the PSW which describes its capabilities.

Registers — Each procedure is provided with its own set of registers which are separate from those of any other procedure.

Parameters — Procedures may have parameters. If a procedure has parameters, then the list of parameters for the procedure is included in the procedure's context.

Exception Handler — Each procedure may define a sequence of code to be executed when an error or abnormal condition is detected. This error code is called an exception handler. The location and state of a procedure's exception handler is recorded in its context.

This collection of information, called the **Procedure Context**, defines the current state of execution of a procedure.

Procedures may call other procedures. This implies that the current procedure context should be preserved and a new procedure context created for the called procedure. Completion of the called procedure should cause its context to be eliminated and the calling procedure's context to be restarted. Thus the call/return behavior of procedures produces a stacking of procedure contexts. In the MCF architecture, the stacked procedure contexts are maintained on a **Context Stack**. Figure 8-1 shows the general form of the context stack.

In each task there is a procedure (usually called the "main program") which has the unique property that it has no caller. The context for this procedure was created by the initiation of execution of the task. A return from this "main program" implies termination of the task. The context of the "main program" forms the BASE of the stack of procedure contexts for this task. This unique procedure context is identified by the fact that its PSW has bit 16, the BASE bit, set. The collection of procedure contexts (the currently active routine plus its caller plus the caller's caller... down to the "main program") is called an **Execution Context.**

8.1. The Context Stacks. There are two active context stacks in the MCF architecture: the Kernel context stack and the Task context stack. These stacks are accessed using one of two context pointers. The context stack used by the executing code is determined by the current PSW. Normal call and return operations do not change the context stack in use. Interrupts, traps, and task completions may cause a context stack switch.

8.1.1. Context Pointers. There shall be two 32 bit registers, called the Kernel Context Pointer and the Task Context Pointer, which contain the addresses of the currently active procedure contexts on the Kernel and Task context stacks, respectively. The Context Pointer currently in use shall be selected by bit 0 of the current PSW. If this bit is clear, the Kernel Context pointer shall be used, if this bit is set the Task Context pointer shall be used. These registers shall be accessible in the I/O space and by the privileged LTASK and STASK instructions. The operation of these instructions insure that the address contained in the context pointer is aligned on a 32 bit word boundary.

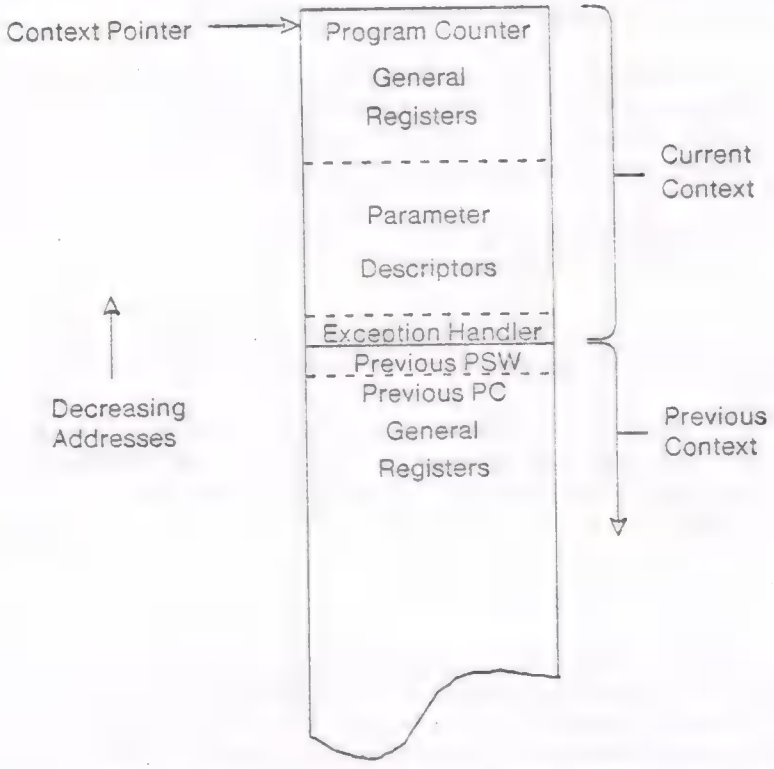


Figure 8-1: Stacked Procedure Contexts

8.1.2. Structure of Context Stacks. The context stack selected by the current context pointer shall be used by the executing code. The context of each executing procedure shall be stored in the following order (increasing addresses):

1. The PSW for this procedure.
2. The registers for this procedure, starting with the program counter (register 0), up to the maximum accessible register as defined in this procedure's PSW.
3. The parameter descriptors for this procedure (defined in section 8.4.4).
4. The state of the exception handler for this procedure (defined in section 9).

The context stack shall expand downward. A call by the currently executing procedure shall cause the context pointer to be decremented by the size of the context of the called procedure. The context of the called procedure shall then be placed in the area defined by the decremented context pointer.

8.1.3. Cacheing of the Context Stack. The context of a procedure contains areas reserved for a variety of frequently accessed quantities such as the PC, PSW, general registers and so on. In many implementations it will be desirable to maintain such information in fast registers. This is particularly true for the currently executing procedure, its program counter and registers. The MCF architecture does NOT define the properties of any such cacheing mechanism. The behavior of the context area of the active (Kernel and Task) context stacks is IMPLEMENTATION DEPENDENT. The value of such memory locations is undefined. The effect of storing into such memory locations is unpredictable. The memory management system provides a mechanism for protecting against such invalid software actions.

8.1.4. Changing of Context Stack Pointers. The MCF architecture provides two active context stacks, corresponding to the Kernel Context Pointer and Task Context Pointer. Since each task in the system has associated with it a unique execution context, a mechanism for changing the active context stack pointer is provided. The LTASK and STASK instructions provide this facility. Additionally, these instructions shall force consistency between the context in memory and any cached state. For a complete description of their functions, refer to the individual instruction descriptions.

8.1.5. Alignment of Context Pointers. The question of which item in the current procedure context is pointed to by the context pointer is implementation dependent. The address in the context pointer shall be greater than or equal to the smallest address occupied by the current context. When a new context is created, the context pointer prior to being decremented shall be greater than the address of any byte of newly created context. These restrictions imply that a context area may be initialized by setting the context pointer to the greatest word address in the context area plus 4.

8.2. Procedure Descriptor. The characteristics of a procedure are specified by a procedure descriptor located at the entry point(s) of the procedure. This 16 bit descriptor will be aligned on a word boundary. The format of the procedure descriptor is shown in figure 8-2. An optional 16 bit displacement follows if an exception handler is defined. On entry to the procedure, the information contained in bits 2:15 is recorded in bits 18:31 of the PSW. Execution of the procedure shall begin at the location immediately following the procedure descriptor and optional displacement halfword located at the addressed entry point. The function of each field of the procedure descriptor is described below.

8.2.1. Fixed/Variable Number of Parameters. Bit 0 of the procedure descriptor shall specify whether the number of parameters for this procedure is fixed or variable. If bit 0 is clear, bits 8:15 will contain the number of parameters. If bit 0 is set, bits 8:15 are reserved and the first operand of the parameter list shall be evaluated as the unsigned integer number of parameters. In either event, the number of parameters for the procedure shall be placed in bits 24:31 of the PSW.

Some procedure entries, such as interrupt and trap handlers, have their number of parameters fixed by the hardware. In such procedure descriptors, bits 0 and 8:15 are reserved.

8.2.2. Exception Handler. An exception handler for a procedure may be predefined. If bit 1 of the procedure descriptor is set, an exception handler shall be defined for this procedure. The address of the exception handler shall be determined by adding the signed 16 bit displacement contained in the halfword following the descriptor to the address of this halfword. If bit 1 is clear, no exception handler shall be defined for this procedure. Section 9.2 explains the semantics of defining an exception handler.

8.2.3. Exception Propagation Control. Bit 2 of the procedure descriptor shall be copied to bit 18 of the PSW. This bit defines the direction of exception propagation. See section 9.4.

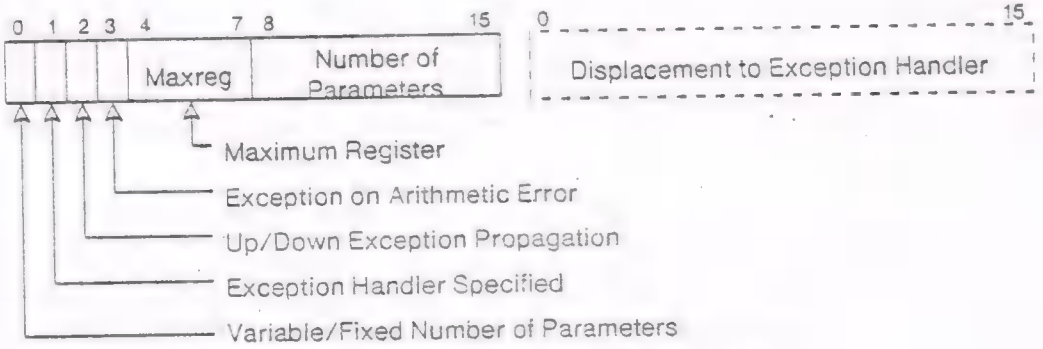


Figure 8-2: Procedure Descriptor

8.2.4. Arithmetic Error Control. Bit 3 of the procedure descriptor shall be copied to bit 19 of the PSW. This bit defines the action to be taken on arithmetic errors. See section 6.14.

8.2.5. Available Registers. Bits 4:7 shall determine the maximum numbered register accessible to this procedure. This value shall be copied to bits 20:23 of the PSW. This number of registers (and a program counter) shall be allocated for use by this procedure.

8.3. Procedure Invocation. In the MCF architecture calls, supervisor calls, illegal instructions, supervisor handled exceptions, task initiations, interrupts, and traps are all handled as procedure invocations using a common mechanism. They differ primarily in the manner in which they determine the entry address of the procedure to be invoked and the parameters to be passed.

The procedure call mechanism is driven by the procedure descriptor located at the entry address of the procedure. Invocation of a procedure shall cause the following actions to occur:

- The current PSW is saved in the context stack and a new PSW is created.
- The register set of the invoked procedure is allocated.
- The exception handler for the procedure is initialized.
- The parameter descriptors for the procedure are initialized.

The current context pointer shall be updated to reflect the new procedure context added to the context stack. Each of these actions is detailed below.

8.3.1. Determination of a New PSW. The contents of the new PSW are determined by the type of invocation (call, trap, etc.) and by the procedure descriptor. Bits 18:31 of the PSW shall be determined by the procedure descriptor as described in section 8.2. Bit 16 (BASE) shall be set by interrupts, traps, and the TINIT and PINIT instructions (task initiation). It shall be cleared otherwise. Bit 17 (Supervisor) shall be set equal to the most significant bit of the procedure entry address. Note that the CALL instruction will produce a trap if a user attempts to set this bit. Bits 0:14 of the PSW shall be set in the following manner:

CALL	Unchanged

SVC	Bits 13:14 are cleared. Bits 0:12 are Unchanged
Supervisor Exception Handler	Bits 13:14 are cleared. Bits 0:12 are Unchanged
Illegal Opcode	Bits 13:14 are cleared. Bits 0:12 are Unchanged
Task Initiation	The TINIT and PINIT instructions specify the contents of bits 0:15 of the PSW as one of their operands.
Traps	Last mode (bit 1) shall be set to the previous value of bit 0. Bit 0 (Kernel/User) shall be clear. This forces the trap to use the Kernel context stack. Bits 2:3 shall be clear. Bits 4:8 shall be unchanged. Bits 9:14 (the condition codes and debug control) shall be clear.
Interrupts	The new PSW shall be determined as for traps except the priority (bits 4:8) shall be set to the priority of the interrupting device.

The privilege bit of the new PSW (bit 15) shall be set equal to bit 31 of the vector for SVC, supervisor exception handler, illegal opcode, trap and interrupt entries. Task initiation specifies this bit explicitly. Call instructions leave the bit unchanged or clear it, depending on the opcode.

8.3.2. Register Set Allocation. The registers of a called procedure are separate from those of the caller. The number of registers available to the invoked procedure shall be determined by the procedure descriptor. Space shall be allocated on the context stack for each register starting with 0 (the program counter) up to the maximum specified in the procedure descriptor. The contents of these newly created registers are undefined with the exception of register 1. If this is not an interrupt or trap entry, and register 1 exists in both the caller and called procedure's registers, then the value of the caller's register 1 shall be copied to the called procedure's register 1. Otherwise, the contents of register 1 are also undefined. With this single exception, there is no inheritance of registers in the MCF architecture.

8.3.3. Definition of Exception Handler. Space in the procedure context shall be provided to contain the state of the exception handler for this procedure. The exception handler shall be set in the Disabled or Handler Available states, as specified by the procedure descriptor (see section 9).

8.3.4. Initialization of Parameter List. The parameters for a procedure invoked by a CALL, SVC, or unimplemented instruction are specified explicitly in a parameter list. The parameters of an interrupt, trap, or supervisor exception handler invocation are implicitly defined by the architecture. Task initiations have no parameters. An area of the procedure context shall be reserved for the parameter descriptors of the invoked procedure. There shall be one such descriptor for each parameter of the procedure. Parameter descriptors are defined functionally in section 8.4.

8.4. Parameter Lists. The passing of arguments or parameters to a called procedure is accomplished by specifying a parameter list. In calls, supervisor calls, and opcode exceptions the parameter list is explicitly specified while in certain interrupts and traps it is implied by the architecture. The specified parameters are accessed using the parameter addressing modes which determine the number of the parameter to be accessed (1st, 2nd, 3rd, etc.). The representation of these addressing modes is described in section 5.8. The semantics of parameter access are defined below.

8.4.1. Parameter Specification. Explicit specification of parameters is accomplished by a parameter list which immediately follows the call instruction. The parameter list shall consist of a sequence of operand specifiers. These operand specifiers may use any of the addressing modes described in section 5. The number of operand specifiers shall be determined in the following manner:

1. The calling instruction (CALL, SVC, or OPEX) will specify the address of the procedure to be called. At this address there will be a procedure descriptor, as described in section 8.2.

2. If bit 0 of the procedure descriptor is clear, bits 8:15 of the procedure descriptor shall specify the number of parameters. If bit 0 is set, the first operand of the parameter list shall be interpreted as the unsigned integer number of parameters. The remaining operands of the reduced parameter list shall be interpreted as the actual parameter list.

Each operand specifier in the parameter list specifies the location and size of one operand. The location of each operand shall be determined using the state of the CALLER at the time of the call. Any registers specified in the parameter list shall refer to the caller's registers; Any addresses in the parameter list shall be computed using the caller's registers as they exist when the call is executed.

8.4.2. Parameter Access. The parameters specified in the parameter list are conceptually numbered in increasing order, starting at 1. The parameter addressing modes specify a parameter number. The appearance of a parameter addressing mode as an operand shall be interpreted as an access to the correspondingly numbered parameter of the current context. Parameter 0 shall be interpreted as the number of parameters accessible in the current context; This information is recorded in bits 24:31 of the PSW. Parameter 0 cannot be written; An attempt to do so shall cause a **Illegal.Write** exception. Bits 24:31 of the PSW specify the maximum parameter number in the current context. An attempt to access a parameter number greater than this shall cause an **Illegal.Parameter** exception.

8.4.3. Example of Parameter Linkage. The context stack generated by this example (assuming 2 registers for both the caller and callee) is shown in figure 8-3. Assume X is a procedure with a fixed number of parameters =3 and that both X and its caller have access to a register 2 (R2). The following call:

```
Call    X, #5,A↑W, R2
```

Would set up the following correspondence between parameter numbers (written as ?n) and parameters in the parameter list:

```
?0          =       #3          ;Number of parameters
?1          =       #5          ;A literal 5
?2          =       A↑W         ;A 32 bit word in memory named A
?3          =       R2          ;The caller's register 2
?4,?5,...           illegal
```

The use of parameter addressing modes in this context would have the following effects:

```
MOV     ?1,R2 <=>       MOV     #5,R2
MOV     ?2,R2 <=>       MOV     A↑W,R2
MOV     ?3,R2 <=>       MOV     R2*,R2
```

* NOTE that the last example moves the contents of the CALLER's register 2 to the called routine's register 2.

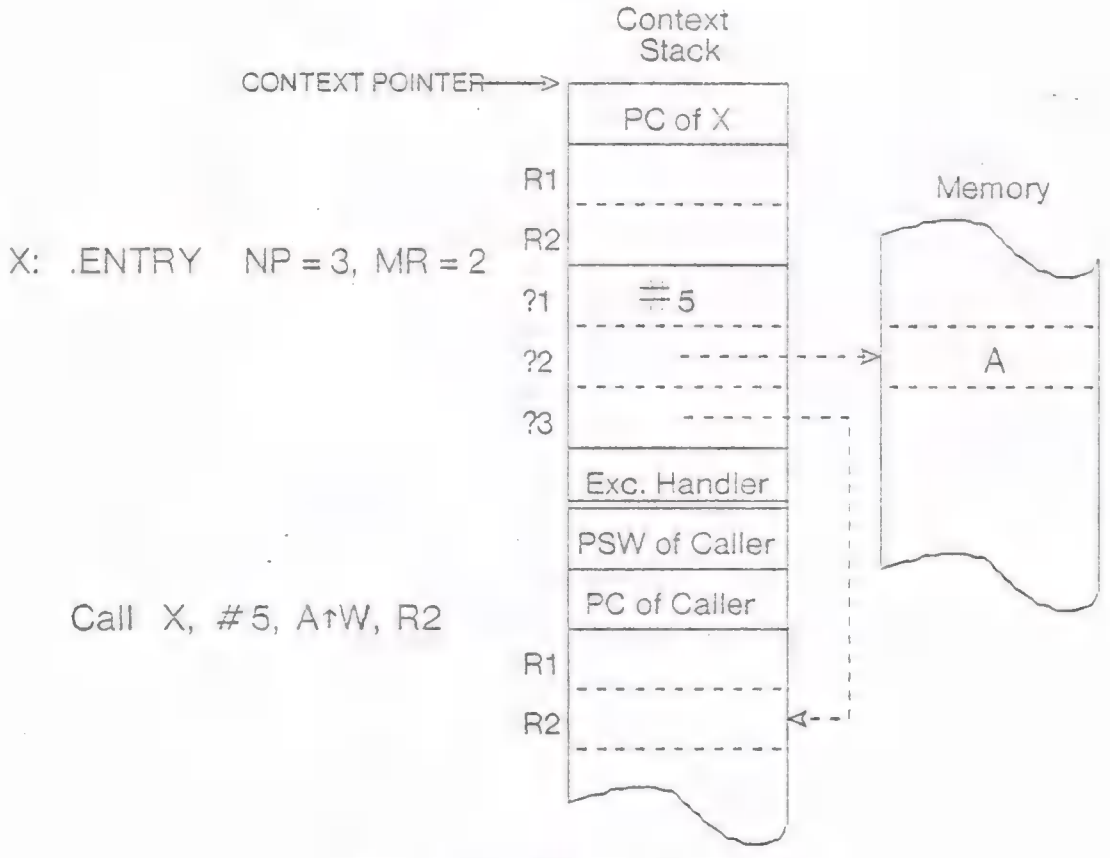


Figure 8-3: Example of a Call Context

8.4.4. Mechanism of Parameter Addressing. The parameter access mechanism can be viewed as a splitting of the operand access mechanism used by all instructions. Figure 8-4 is a conceptual picture of the MCF microengine. Operand specifiers are reduced to location and size information by operand pre-evaluation. This primitive information is used for operand access during operand evaluation.

The operands of parameter lists, however, are not fully evaluated in this manner. Instead, the location and size information from pre-evaluation is diverted to the appropriate parameter descriptor in the context stack as shown in figure 8-5. This reduces the parameter list to a linear descriptor array.

Access to parameters is accomplished by parameter addressing modes. These operand specifiers evaluate to an unsigned integer parameter number. This index is used to recover the position and size information corresponding to the specified parameter. The mechanism is diagrammed in figure 8-6.

The size and format of the parameter descriptors is implementation dependent. The encoding of these descriptors is constrained only by the functional requirements placed on the parameter passing and access facilities. The following is an example of an encoding sufficient to meet these requirements. In this example the parameter descriptors specify one of four possible locations for the

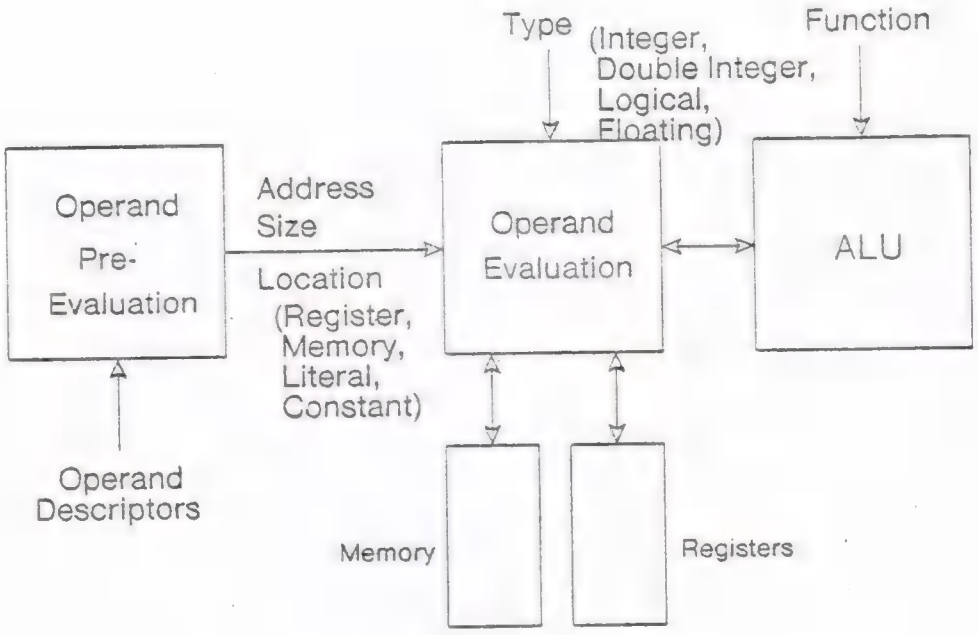


Figure 8-4: Normal Operand Processing

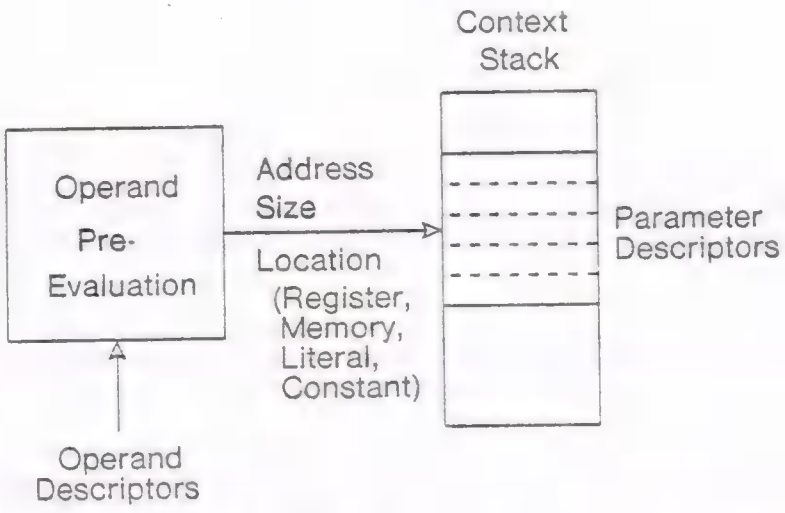


Figure 8-5: Parameter List Operand Processing

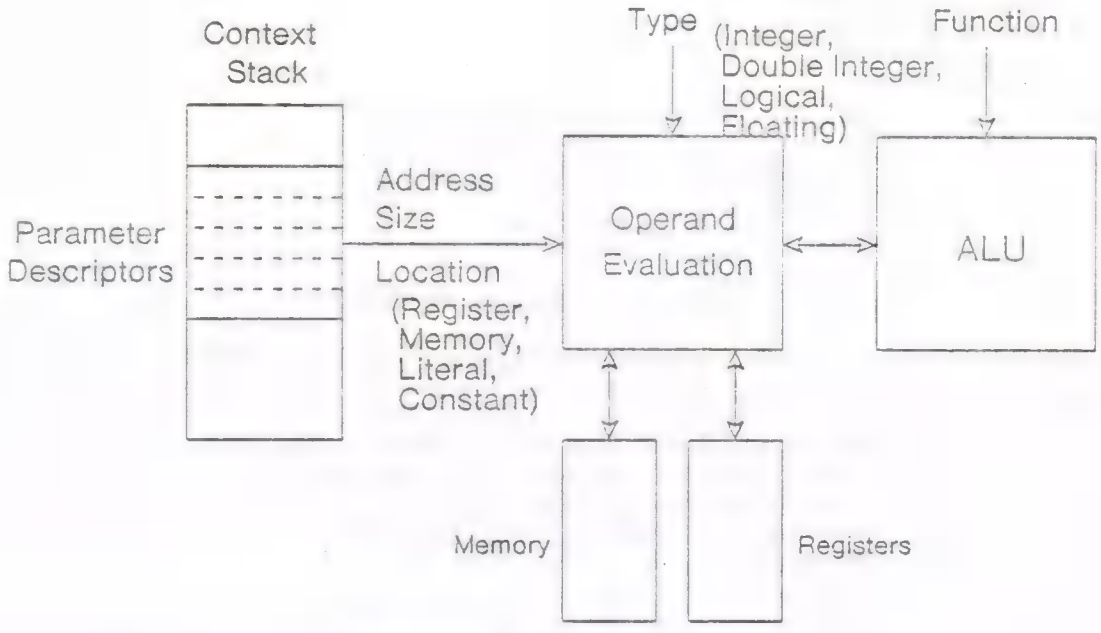


Figure 8-6: Parameter Access

parameter:

Constant — The parameter is a value contained in the parameter descriptor. Such a parameter is read-only. An attempt to write a constant parameter will cause an **Illegal.Write** exception.

Literal — The parameter is a literal contained in the instruction stream. The address and size of the literal are encoded in this descriptor. This parameter is read-only. An attempt to write this parameter will cause an **Illegal.Write** exception.

Register — The parameter is a register in the context stack. The size of the parameter is 32 bits. A pointer to the register in the context stack is encoded in this descriptor.

Memory — The parameter is a memory location. The size and address of this location are encoded in this descriptor.

8.5. Vectored Calls: SVC and OPEX. The supervisor call and unimplemented opcode (OPcode EXception or OPEX) facilities utilize a vectoring mechanism controlled by the supervisor. Vectoring shall be controlled by a pair of processor registers for SVCs and a pair for OPEXs. The format of these registers is shown in figure 8-7

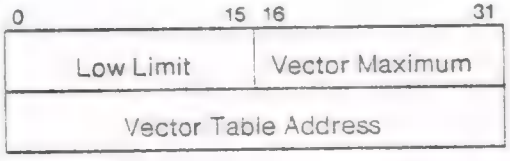


Figure 8-7: SVC and OPEX Vector Registers

The vectored instruction shall select the appropriate register pair (SVC or OPEX) and specify an index. This index shall be compared as an unsigned quantity with the low limit specified in the first register of the pair. If the index is less than the limit, or if (index - low limit) is greater than vector max, the 32 bit word located at the virtual address in the second register (vector table address) shall be used as the vector. If the index is within the specified range, the quantity vector table address + (index - low limit)*4 shall be used as the virtual address of the vector. The contents of the 32 bit vector shall be used as the entry address of the procedure. Access to this vector shall be allowed regardless of the state of the privilege bit of the PSW. The vectoring operation is shown in figure 8-8.

8.6. Vector Format. All vectored operations in the MCF architecture (interrupts, traps, SVCs, OPEXs and supervisor exception handler entries) use a common vector format. Vectors are 32 bit words with the format shown in figure 8-9. Bits 0:29 of the vector specify bits 0:29 of the entry address. Bits 30:31 are assumed zero. Bit 31 of the vector is copied to bit 15 of the new PSW (privilege).

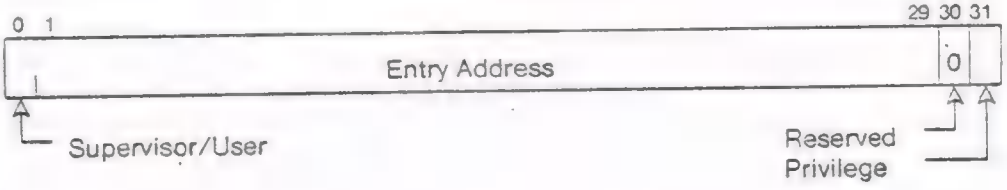


Figure 8-9: Vector Format

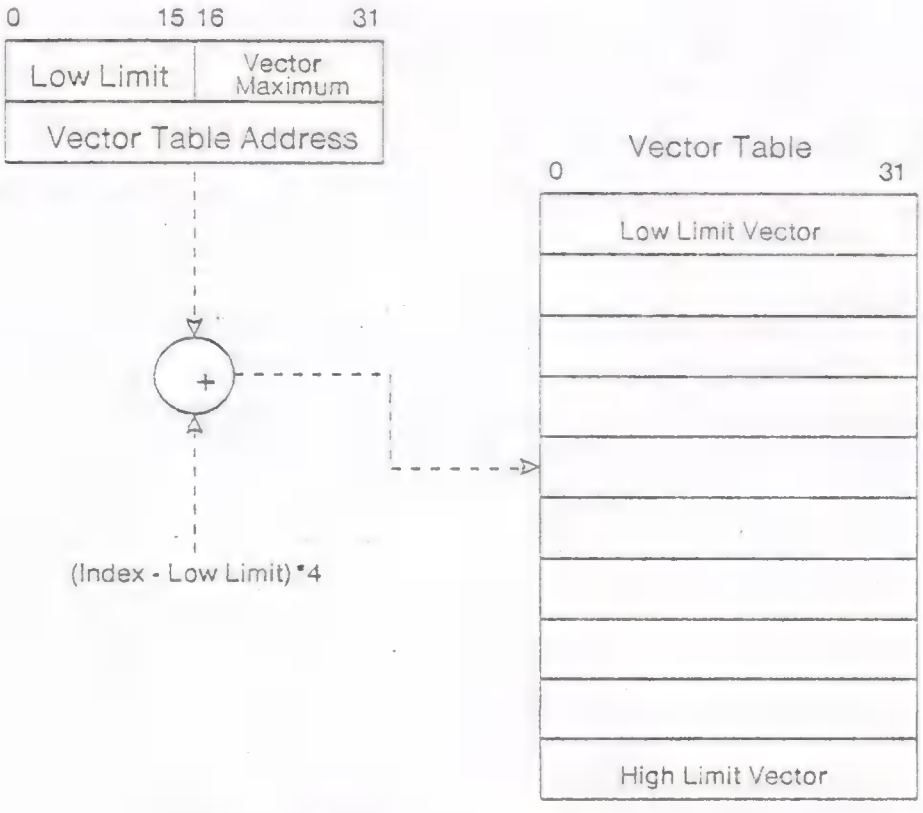


Figure 8-8: Vectored Call Address Calculation

9. Exceptions

Program errors are handled by the exception facility. Exceptions may be handled by the procedure in which they occur, or by the supervisor, or by the caller of the procedure in which they occur.

9.1. Procedure Exception Handler. There shall be an entry in the context of each procedure defining the state of the current exception handler associated with that procedure. The exception handler of a procedure shall be in one of three states:

Disabled — No exception handler is currently defined for this procedure.

Handler Defined — The starting address of the exception handler for this procedure is recorded in the context stack.

Exception Code Available — No exception handler is currently defined for this procedure. The exception code for the last exception is recorded in the context stack.

The states of the exception handler for a procedure, and the allowed transitions, are shown in figure 9-1. Encodings of the states in the context area shall be implementation dependent.

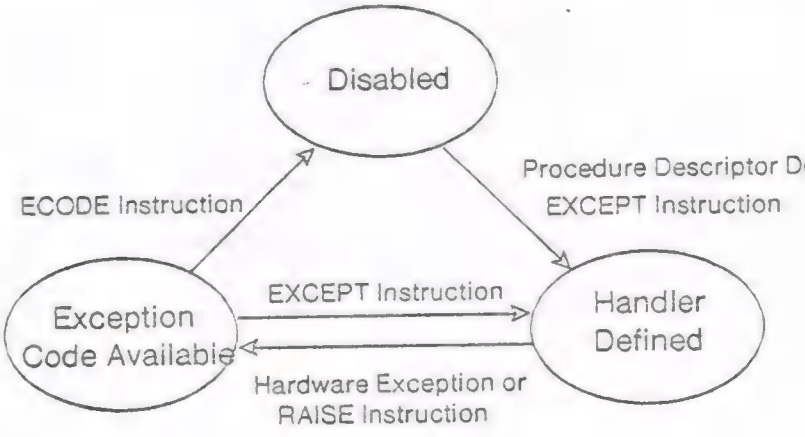


Figure 9-1: States of an Exception Handler

9.2. Definition of an Exception Handler. The exception handler associated with a procedure can be defined by two means. If bit 1 of the procedure descriptor is set on entry to a procedure, the exception handler for the procedure shall be defined to start at the address specified. Refer to section 8.2. If bit 1 is clear, the exception handler shall be set in the disabled state. The second means of defining an exception handler is by use of the EXCEPT instruction. When EXCEPT is executed, the exception handler for the current procedure shall be defined to start at the address specified by the EXCEPT instruction.

9.3. Raising an Exception. An exception may be raised by the RAISE or ERET instructions, or by the detection of an abnormal condition by the hardware. An exception code is associated with each exception raised. The exception code is explicitly specified by the RAISE and ERET instructions as an operand. The exception code specified shall be treated as an unsigned integer of at least 24 bits. Hardware generated exceptions shall have associated with them the following fixed codes:

Exception Name	Code (Decimal)	Described in Section
Specification.Error	1	23
Illegal.Mode	2	5.2
Illegal.Parameter	3	8.4.2
Illegal.Register	4	5.1, 6.15
Illegal.Write	5	5.5, 5.6, 8.4.2
Illegal.Size	6	27
Illegal.Address	7	5.3
Bad.Displacement	8	21
Context.Alignment	9	23
Context.Base	10	28
Illegal.Divisor	16	16.2
Truncation	17	16.2
Range.Error	18	20
Illegal.Operation	19	29.3.1
Divide.By.Zero	20	29.3.2
Floating.Overflow	21	29.3.4
Floating.Underflow	22	29.3.3
Floating.Inexact	23	29.3.5
Unordered	24	29.6
Task.Failure	32	9.4, 24
Break	33	28
Instruction.Break	34	10
Call.Break	35	10

9.4. Exception Handling. The control transfer on an exception is diagrammed in figure 9-2. When an exception is raised, the action which occurs shall depend upon the Up/Down Level Exception bit, bit 18 of the PSW. If UDLE is set, the state of the current procedure's exception handler shall be unaffected, and the Supervisor Exception Handler shall be invoked (described below). If UDLE is clear, the action taken shall depend on the state of the current context's exception handler. If the current state is Handler Defined, the program counter shall be set to the starting address specified, the state shall be changed to Exception Code Available, and the exception code shall be recorded in the current context area. If the current state is not Handler Defined, the action taken shall depend on the Base bit (bit 16) of the PSW. If this bit is clear, the current context shall be removed from the context stack and the exception shall be re-raised in the caller's context. If the Base bit is set, there is no caller. The current context shall be removed from the context stack and the Supervisor Exception Handler shall be invoked with the Task.Failure exception. The BASE bit in the PSW of the supervisor exception handler shall be set.

9.5. Supervisor Exception Handler. The supervisor exception handler is a procedure defined by the supervisor to which control may be transferred on occurrence of an exception. This facility is enabled at the procedure level by the UDLE bit in the procedure descriptor. The entry address of the supervisor exception handler is specified in a single word vector at physical address 24 (Hex). The supervisor exception handler shall be invoked as a procedure using the current context stack as

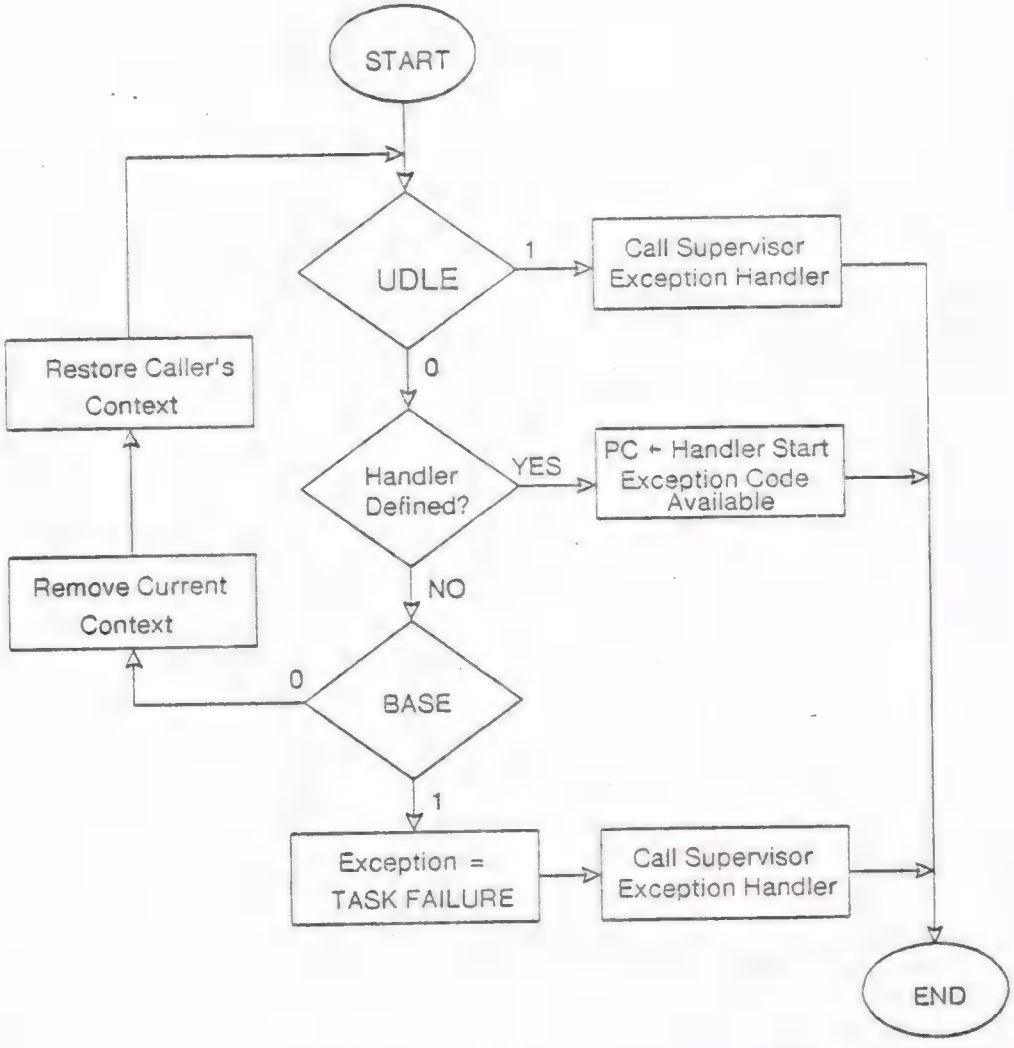


Figure 9-2: Transfer to an Exception Handler

described in section 8.3. There shall be three parameters to this procedure implicitly defined by the architecture:

?1 The exception code associated with the exception. This parameter shall be a literal constant; it cannot be written.

?2 The program counter at the beginning of the instruction causing the exception, as the address of a byte in memory. Access to this parameter is equivalent to using indirect register on the original program counter to fetch a byte: @OLDPC+B.

?3 The program counter of the context which invoked the supervisor exception

handler, as a register. This program counter contains the address of the instruction to be executed if the supervisor exception handler does a RET. This parameter may be read or written. Note that due to propagation of the exception this may not be the program counter of the original offender. If this is a **Task.Failure** situation, this parameter is not defined.

Thus transfer to the supervisor exception handler looks like a procedure call with the exception code and offending instruction as parameters. The supervisor exception handler may choose to take corrective action, abort the task, return the exception to the offending procedure (using ERP), or simply continue (using RET).

10. Debugging Facilities

The MCF architecture provides the ability to monitor the execution of a program through tracing and break facilities at the instruction and procedure levels. These facilities operate through the supervisor exception handler mechanism which serves as the interface between the debugger and the rest of the system. Control may be transferred to the supervisor exception handler either by explicit use of the BREAK instruction or implicitly by the program trace facility.

10.1. Program Tracing. Bits 13:14 of the PSW control the program trace facility. The defined settings of these bits are:

00	Disabled
01	Instruction Break. A break shall be generated after each instruction execution with exception code Instruction.Break.
10	Call Break. A break shall be generated after each CALL or CALLU instruction execution with exception code Call.Break

The breaks specified by the above codes shall occur after the execution of the specified instruction and before a check for pending interrupts. An instruction which causes bits 13:14 of the PSW to be set to 01 shall cause a break at the end of its execution. A break shall not occur after an instruction which clears these bits.

The generation of a program break shall cause an entry into the Supervisor Exception Handler as described in section 9.5. The exception code parameter shall be set as described above.

The semantics of the procedure call mechanism imply that the setting of the trace bits (13:14 of the PSW) are propagated into called routines. Thus enabling trace in a procedure will also cause tracing of all procedures it explicitly calls. The trace setting is not propagated into SVCs, CPEXs, interrupts or traps. Thus the tracing is confined to the task in which it is initiated. Execution of an RET (or ERET or ERP) causes the trace bits to be restored to the value defined by the saved PSW. Return from a procedure will therefore cause tracing to resume as specified regardless of the conditions existing in the returning procedure.

11. Interrupts and Traps

Interrupts and traps are events requiring a change in the execution environment in the processor. Interrupts are asynchronous events generated externally or independently of the executing instructions: Traps are conditions caused by the executing instructions. In the MCF architecture interrupts and traps are treated as parameterized calls. The entry address for such a call is determined by a vector in physical memory. The interrupts and traps supported in the MCF architecture are described below.

11.1. Interrupt Priority. Acceptance of interrupts shall be on a priority basis. The current priority of the processor shall be specified by bits 4:8 of the PSW. Priority shall be encoded as an unsigned integer with 0 being the lowest priority and 31 the highest priority. Power Fail/Restore Interrupts have the highest priority and cannot be locked out. I/O and software interrupt requests may be made at any priority between 1 and 31. The highest priority interrupt request with priority greater than the processor's shall be accepted. Interrupts with priority less than or equal to the processor's shall not be accepted. In the event of an I/O and a software interrupt request with equal priority, the I/O request shall be accepted first. The order of acceptance of I/O interrupt requests of equal priority is implementation dependent.

11.2. I/O Interrupts. I/O interrupts are requested by a device or controller independent of the processor. The interrupting device will specify a priority and a physical vector address. Acceptance of an interrupt shall cause the 32 bit vector located at this address to be examined. If the most significant bit of the vector (bit 0) is set, the vector shall be used as the entry address of the procedure to be invoked. This procedure shall be invoked on the Kernel Context stack as described in section 8.3. If bit 0 of the vector is clear, then the contents of the 32 bit vector at physical address 8 (Hex) shall specify the procedure entry address. The procedure shall be invoked with a single parameter: the contents of the device specified vector as an address length constant. In either case, the priority of the processor (bits 4:8 of the PSW) shall be set equal to the priority of the interrupt request. Figure 11-1 is a diagram of the I/O interrupt sequence.

11.3. Software Interrupt Requests. An interrupt can be requested by the software using the Software Interrupt Request Register. This 32 bit register contains one bit for each priority level. Setting bit n shall cause a software interrupt to be requested at priority level n. Bit 0 is reserved. Acceptance of a software interrupt request at a given priority level shall cause the corresponding bit in the request register to be cleared. The vector at physical address 4 shall specify the entry address of the procedure to be invoked. This procedure shall be invoked on the Kernel Context Stack at the requested priority level as described in section 8.3. The procedure shall have one parameter: a constant equal to the priority level of the software interrupt.

11.4. Power Failure Interrupt. Detection of a loss of power condition shall cause the following actions to occur:

1. The 32 bit word at physical address 20 (Hex) will contain the physical address of a two word power fail save area. The hardware shall store in this save area the current contents of the Kernel Context Pointer, the Supervisor Map Pointer, and the Kernel bit (bit 0) of the current PSW.

2. The context stacks in memory shall be updated to reflect the contents of any implementation dependent context caches.

3. The procedure whose entry address is contained in the word at physical address 18 (Hex) shall be invoked with no parameters on the Kernel context stack at priority 1F (Hex).

The power fail routine invoked by this mechanism may then save any other volatile information

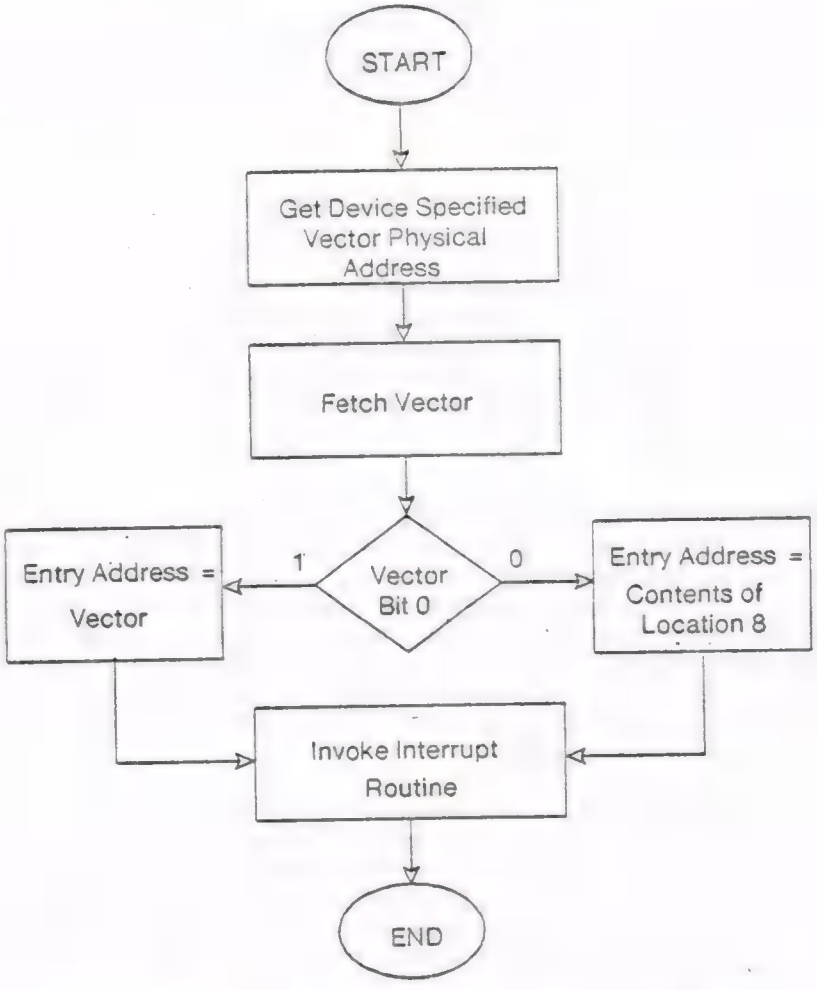


Figure 11-1: Vectoring of I/O Interrupts

needed to restart execution on power up. Note that the Kernel context was saved before the power fail routine was invoked. Therefore, the context of the power fail routine will be lost in the power fail.

11.5. Power Restore Interrupt. If power is applied to the processor with the Run switch enabled and the contents of memory intact, the following actions shall occur:

1. The Software Interrupt Request Register and the Auxiliary Status Register shall be cleared.

2. The Kernel Context pointer, the Supervisor Map Pointer, and the Kernel bit (bit 0 of the PSW) shall be restored from the power fail save area specified above.

3. Any implementation dependent caches of the Kernel context stack or supervisor memory

map shall be made consistent with those stored in memory.

4. The procedure whose entry address is contained in the word at physical address 1C (Hex) shall be invoked. This entry shall use the Kernel Context stack with no parameters at priority 1F (Hex) as an interrupt entry. Note that the Kernel bit restored in step 1 is used to set the last mode bit in this procedure's PSW. Return from this procedure will therefore cause the appropriate context stack to be reactivated.

The contents of all other processor registers are unpredictable. Restoration of these registers is the responsibility of the power restore routine.

11.6. Memory System Error Traps. The MCF architecture provides support for detection of memory failures such as parity errors. There are two types of memory errors defined:

Soft Errors	This is an informational signal indicating that an error occured within the memory system that was corrected by the memory system. The data transfer was completed correctly. For example, an incorrect bit read from memory might be corrected by a Hamming code mechanism.
Hard Errors	An uncorrectable error was detected in a memory data transfer. The information transferred is probably wrong.

The ability to detect either type of error is implementation dependent. If this ability is provided, it shall function as specified below.

11.6.1. Hard Memory Errors. Hard memory errors shall cause a trap using the Kernel context stack. The address of the procedure entry invoked shall be specified by the vector at physical address 14 (Hex). The procedure shall be invoked with a single parameter which is a constant specifying the physical address which failed.

11.6.2. Soft Memory Errors. Soft memory errors may be masked. This allows the processor to be informed of such errors without being saturated by repeated reports. If bit 0 of the auxiliary status register (figure 7-1) is clear, soft errors shall be ignored. If this bit is set, it shall be cleared and a trap shall occur using the Kernel context stack. The address of the procedure entry shall be specified by the vector at physical address 10 (Hex). The procedure shall be invoked with a single parameter which is an address length constant specifying the physical address which failed.

11.7. Privileged Instruction Trap. An attempt to execute a privileged instruction with bit 15 of the PSW clear shall cause a trap using the Kernel context stack. The address of the procedure entry to be invoked shall be specified by the vector at physical address 28 (Hex). The procedure shall be invoked with a single parameter. The parameter shall be the program counter at the beginning of the instruction, as a byte in memory. Access to this parameter is equivalent to using indirect register mode on the original program counter to fetch a byte : @OLDPC↑B.

11.8. Memory Management Traps. A memory management trap is generated by an attempt to perform a memory access which is specified as invalid by the memory management facility (see section 12). Such an access shall abort the instruction and cause a trap using the Kernel Context stack. The address of the procedure entry to be invoked shall be specified by the vector at physical address 0C (Hex). The procedure shall be invoked with four parameters as described in section 12.4.

12. Memory Management System

The memory management system performs the following functions:

- Translation of virtual addresses to physical addresses
- Separation of areas of the virtual address space according to allowed accesses
- Separation of physical addresses into memory and I/O space addresses
- Maintenance of access information

These functions are controlled by memory maps stored in physical memory which are pointed to by map pointer registers in the processor. This relationship is shown in figure 12-1.

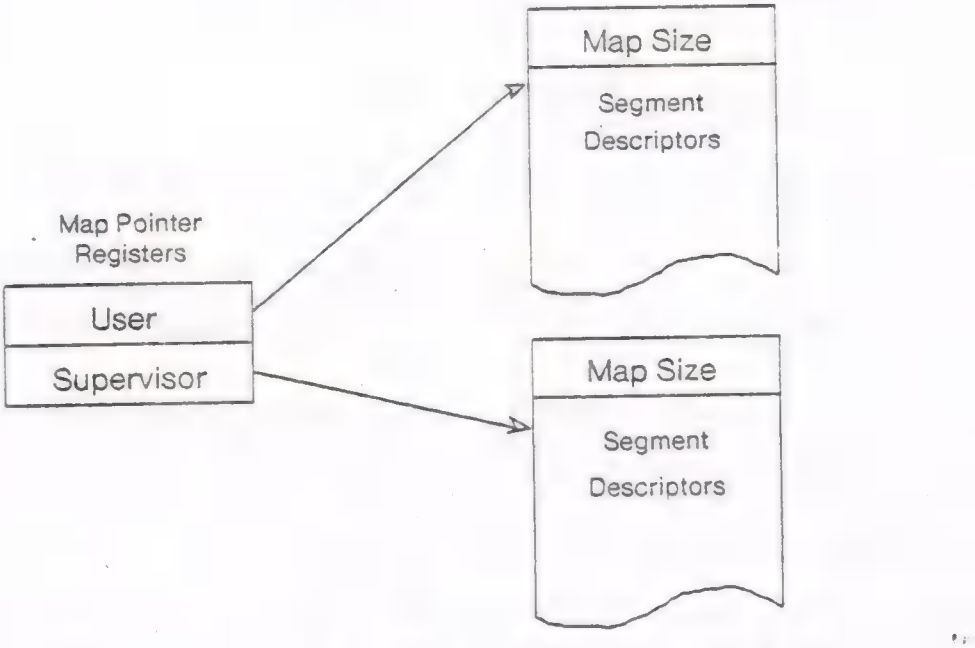


Figure 12-1: Map Data Structure

12.1. Virtual Address Space. The size of the virtual address space is defined as the number of distinct byte virtual addresses which can be generated by the processor. In the MCF architecture two distinct measures of the virtual address space are defined:

- The Architectural Virtual Address Space is the number of distinct byte virtual addresses which can be generated by the addressing modes of the architecture. These addressing calculations are visible to the programmer through the MOVA (move address) instruction. The architectural virtual address space shall be 2^{32} bytes.
- The Implementation Virtual Address space is the minimum of the number of distinct byte virtual addresses which can be generated and mapped using an addressing mode with

memory mapping enabled. In the MCF architecture this may be less than the Architectural V.A.S. due to limitations in the parameter descriptor and mapping mechanisms. The implementation virtual address space shall be a minimum of 2^{24} bytes or 8 times the maximum implemented physical address space, whichever is greater.

These restrictions are designed to allow reasonable optimization of the parameter and memory map implementation while avoiding any software dependencies on these implementation dependencies. The manner in which the software manipulates addresses is driven by the architectural v.a.s, which has been specified as the full 32 bits. The implementation v.a.s impacts the allocation of address space. This is specified to exceed the point at which other software efficiency considerations conspire to limit allocation.

12.2. Mapping of Virtual Addresses. The virtual address space is divided into two halves, one of which is accessible only to supervisor programs. Virtual addresses with the most significant bit (bit 0) set are mapped through the supervisor memory map. Virtual addresses with the most significant bit clear are mapped through the User memory map.

12.2.1. Map Pointer Registers. There shall be two map pointer registers: One for Supervisor and one for User state. The map address register used to translate a virtual address shall depend on bit 0 of the virtual address. If bit 0 is clear, the User Map Pointer register shall be used. If bit 0 is set, the Supervisor Map Pointer register shall be used. The format of these registers is shown in figure 12-2. These registers contain the physical address in memory of the memory maps. Bits 29:31 of these addresses shall be assumed 0. The address in these registers shall be the address of the first map entry in the memory map. Bits 30:31 control the translation of addresses using the selected map pointer register. These bits shall function as follows:

00 — No relocation or protection. Bits 1:31 of the virtual address shall be used as a physical address. I/O space accesses shall be segregated as described in section 12.5.

01 — Protection enabled. Accesses shall be checked for validity as described below. No relocation is performed. Physical addresses are determined as above.

10 — Reserved. If this code is specified in the map pointer to be loaded by a LTASK instruction, a Specification.Error exception shall be raised.

11 — Relocation and Protection. Addresses shall be checked and relocated as described below.

12.2.2. Memory Map Structure. The memory map pointed to by a map pointer register shall be assumed to have the following format. The mapping information for each segment is stored in a 64 bit double word with the format shown in figure 12-3. These map entries are stored in consecutive double words, starting with the entry for segment zero, and aligned on double word boundaries. The number of entries in the map is stored in a 32 bit word preceding the segment zero entry. The maximum number of segments allowed is at least 16. The actual number allowed is implementation dependent. The address in the map pointer register is the address of the segment zero entry.

12.2.3. Segment Association. The relocation and protection functions of the memory management system operate by associating a virtual address with one of the segments specified by the map entries. This association is based on the virtual address bound field (bits 1:28) of the map entries. A virtual address is contained in a particular segment if:

1. Bits 1:28 of the virtual address are less than or equal to the virtual address bound of the segment.

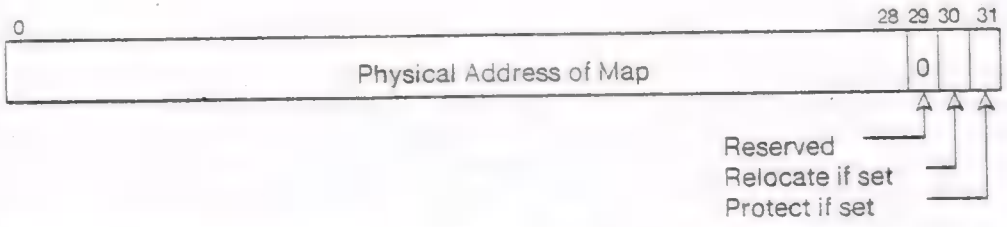


Figure 12-2: Memory Map Pointer Registers

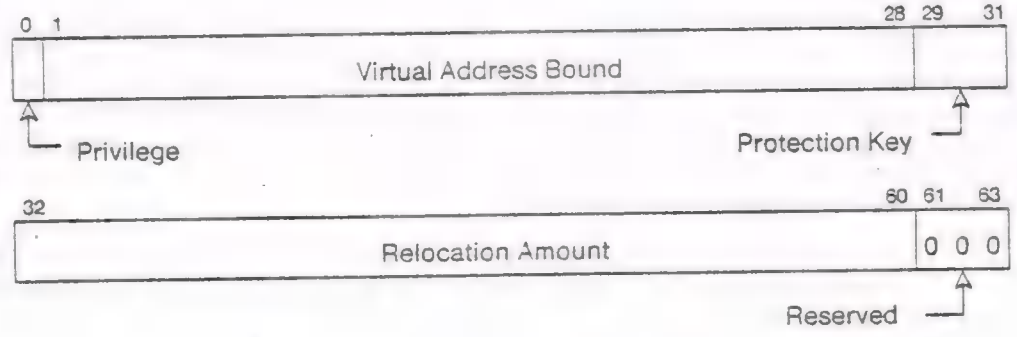


Figure 12-3: Map Entry Format

2. Bits 1:28 of the virtual address are greater than the virtual address bound contained in the previous map entry. This condition shall be satisfied vacuously for map entry 0.

A virtual address for which the above conditions are not satisfied by any map entry shall cause a memory management trap with fault code **Invalid.Segment**.

The map entries will be ordered such that the virtual address bound in each map entry is greater than the bound in the previous map entry. Violation of this restriction by the operating system may produce unpredictable results.

12.2.4. Relocation of Virtual Addresses. If bit 30 of the selected map pointer register is set, relocation is enabled. The virtual address shall be associated with a map entry as described above. Bits 1:28 of the virtual address shall be added to the relocation field of the map entry to form bits 0:28 of the physical address. Bits 29:31 of the physical address shall be equal to bits 29:31 of the virtual address. This operation is shown in figure 12-4.

12.2.5. Access Protection. If bit 31 of the selected map pointer register is set, protection is enabled. The virtual address shall be associated with a map entry as described above. Bit 0 of the map entry specifies whether access to this segment is a privileged operation. If this bit is set, and bit 15 of the PSW is clear, a memory management trap shall be generated with fault code **Privilege.Violation**. Bits 29:31 of the map entry specify the access types allowed for this segment. The encoding of this field is:

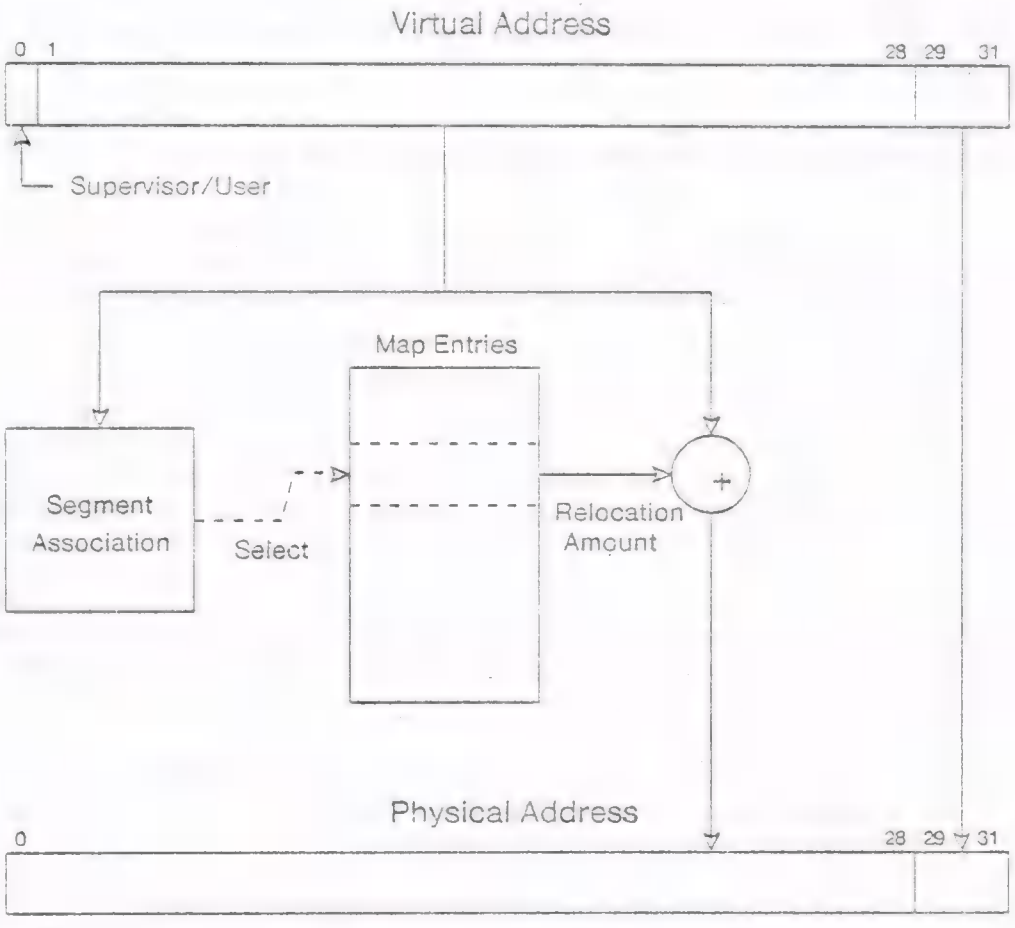


Figure 12-4: Address Relocation

000	No Access. Any access shall cause a trap.
100	Instruction Access Only. Instruction fetch and reading of literals in short literal or literal operands is allowed.
001	Data Read Only. Reading of operands is allowed.
101	Instruction or Read Access. Instruction fetch, literal fetch and reading of operands is allowed.
010	Data Read/Write. Reading or writing of operands is allowed.
110	Reserved. Any access shall cause a trap.
011	Context Only. Access as part of a context stack is allowed. Any other access shall

cause a trap.

111 Reserved. Any access shall cause a trap.

In all cases accesses other than those explicitly allowed shall cause an Invalid.Access trap (see section 12.4 below). Note in particular that context stacks can only be allocated in segments with access code 011. Such segments cannot be manipulated explicitly by any instruction. This is vital to the security of the system and the transportability of the software.

12.2.6. Protection of the Supervisor. Access to the supervisor map pointer is a privileged operation. A virtual address with bit 0 set shall cause an Invalid.Supervisor trap if bit 17 of the PSW is clear.

12.3. Cacheing of Memory Maps. In many implementations it will be desirable to cache parts of the memory maps, such as the map size and a few recently used map entries. The properties of any such cacheing mechanism are implementation dependant. However, the following consistency requirements shall be met:

- Setting a Map Pointer Register using the LTASK instruction shall force the cache to be consistent with the specified map.
- Alteration of a map entry using the REPENT instruction shall force the cache to be consistent with the new entry and the map size of the altered map, as specified in memory.

12.4. Memory Management Traps. A memory management trap is generated by an attempt to perform a memory access which is specified as invalid by the memory management facility. Such an access shall abort the instruction and cause a trap using the Kernel Context stack. The address of the procedure entry to be invoked shall be specified by the vector at physical address 0C (Hex). The procedure shall be invoked with four parameters:

?1 The virtual address causing the trap, as a byte in memory. Access to this parameter is equivalent to using indirect register mode on the virtual address to access a byte.

?2 The program counter at the start of the instruction causing the trap. This parameter shall be specified as a byte in memory. Access to this parameter is equivalent to using indirect register on the original program counter to access a byte.

?3 The segment number of the segment containing the invalid virtual address as a constant.

?4 The memory management fault code as a constant.

The following memory management fault codes are defined:

Name	Code
Invalid.Supervisor	1
Invalid.segment	2
Invalid.Access	3
Privilege.Violation	4

12.5. I/O Space Selection. Communication with I/O devices in the MCF architecture is accomplished with memory access. A portion of the physical address space is reserved for I/O device registers. Accesses to these addresses transfer information to device control registers. The division of the physical address space into I/O space and memory shall depend on whether translation is enabled. If translation is enabled by the selected map pointer register then virtual addresses FFF00000 to FFFFFFFF (Hex) shall select I/O space. If translation is disabled then the top 2^{20} virtual byte addresses of the implementation virtual address space shall select I/O space. Software can uniformly access I/O space as addresses FFF00000 to FFFFFFFF.

12.6. Subsetting of Memory Management. A particular implementation may subset the functionality of the memory management system in three ways:

1. The size of the virtual addresses translated may be reduced as specified above under implementation virtual address space.

2. The size of the physical addresses generated may be reduced to correspond to the physical memory accessable to the implementation. However, the requirements of section 12.5 shall be met.

3. The segment association function may be reduced. The full architecture allows segment bounds to be specified on any double word boundary. The alignment requirement may be increased up to 256 byte boundaries. If this is done, the unused low order bits of the virtual address bound field of the map entries shall be reserved. The relocation amount field shall be reduced by the same number of bits; these bits shall also be reserved. The number of low order bits of the virtual address which are not changed by the relocation process shall be increased by the same number of bits.

13. I/O controllers

Interaction with I/O devices in the MCF architecture is provided by the ability to access control registers as locations in the I/O portion of the physical address space. Some implementations will provide I/O through a fixed set of ports or channels. This section defines a standard I/O controller architecture for handling transactions on such a channel. The I/O controller is a processor with a limited instruction set (described below). This instruction set allows the IOC to perform predefined transfer, sequencing, and control functions without processor intervention. I/O transactions between the processor and IOC are structured as messages containing the specifics of the transfer requested. The IOC instruction set is designed to process such messages in a general manner and to provide additional intelligence to devices connected to its I/O channel.

13.1. Channel Configuration Registers. Associated with each channel shall be a channel configuration register. This register shall be addressable by the processor as a 32 bit word in the I/O address space. The bits in this register shall be used to specify channel dependent parameters (such as parity or bit rate) for the associated interface. Bits 27:31 of this register shall specify the interrupt priority for IOC error interrupts from this channel. The assignment of the other bits in this register is channel dependent.

13.2. IOC Programs. The operation of each channel is controlled by one or more channel programs. The execution of channel programs on different channels shall appear parallel. Some channels may have more than one channel program associated with them. The sequence of execution of channel programs on the same channel is channel dependent (refer to section 13.7).

Each channel program has a channel program counter and a message pointer register associated with it. These registers are described below. The channel program may perform transfer operations using the data in the record specified by the message pointer register as parameters of the transfer. The channel program may interrupt the processor as required, and return information about the requested transfer in the message.

13.3. Program Counter. The channel program counter (CHPC) is a 32 bit register which contains the address of the next instruction in the channel program. This register is visible in the processor I/O address space. A channel program is specified by loading an address into this register; loading this register by the processor shall also cause all activity in the channel and its associated programs to halt. Channel program counters are restricted to even addresses. If an odd address is loaded into a channel program counter an IOC error interrupt shall be generated with fault code **Program.Alignment**. Refer to section 13.8.1.

13.4. Message Pointer Register. The message pointer register (MP) is a 32 bit register which specifies the address of the current message being processed. This register is visible in the processor I/O address space. Loading this register by the processor shall cause the channel program to begin execution. The value in this register must be even. Loading an odd value into a message pointer register shall cause an IOC error interrupt with fault code **Message.Alignment**. Refer to section 13.8.1.

13.5. Instruction Execution. IOC instructions are represented as one or more halfwords aligned on halfword boundaries. On an active channel, the channel program counter shall contain the physical address of the next instruction to be executed. The program counter shall be incremented by the number of bytes in each instruction fetched.

13.6. Operand Accessing. Instruction operands may be located either in the instruction stream or in the message specified by the message pointer register. The location of each operand is implicitly defined by the opcode. ccess to an operand in the instruction stream shall cause the channel program counter to be incremented by the number of bytes in the operand. Operands in the message shall be selected by a displacement field in the instruction. Twice this displacement added to

the contents of the message pointer register shall be the physical address of the operand.

13.7. Channel Program Sequencing. The number of channel programs associated with a channel and the method of sequencing their execution is channel dependent. The structure and sequencing is defined below for various channel types:

High Speed Interface
: There shall be two channel programs: a master and a slave. In channel to channel mode, the master channel program shall be run if the IOC is currently in control of the channel; The slave program shall be run if the other interface is in control of the channel. Exchange of channel control may occur when a LMP, LMPL, or HALT instruction is executed. The controller responsible for arbitration of channel control is determined at configuration time. If the channel is not operating in channel to channel mode, the slave channel program shall be inactive.

MIL-STD-1553B
: There shall be one channel program per 1553 interface.

RS-449
: There shall be two channel programs: an input program and an output program. They shall execute in apparent parallel. Transfer requests by the input channel shall have priority.

13.8. IOC Interrupts. The IOC shall have the ability to interrupt the processor. Processor interrupts may be initiated by IOC detected errors or by execution of an IOC interrupt instruction. An IOC interrupt shall invoke a service routine at the address specified by the contents of the channel interrupt vector. The routine shall be invoked with two parameters:

?1
: The contents of the Message Pointer Register as a constant. This parameter is read-only.

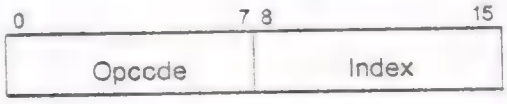
?2
: An interruption code identifying the nature of the interrupt. This is a constant specified by the IOC interrupt instruction or implicitly determined by the nature of the IOC error. This parameter is read-only.

The priority of the interrupt request is specified by the IOC interrupt instruction. The interrupt priority for IOC errors is defined by the channel configuration register for this channel.

13.8.1. IOC Error Interrupts. Detection of certain errors by the IOC shall cause an IOC Error Interrupt to be generated. The channel program executing at the time the error is detected shall be halted. Its message pointer shall be a parameter of the interrupt as described above. The interruption code shall be fixed by the type of error:

Fault	Fault Code (Hex)	Described in Section
Program.Alignment	8001	13.3
Message.Alignment	8002	13.4
Illegal.Opcode	8003	13.9
Transfer.Unit	8004	13.9.1
List.Alignment	8005	13.9.1.1

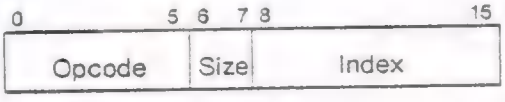
13.9. IOC Instructions. IOC instructions have the general form shown below: The index field is used as a halfword displacement from the message pointer register or the channel program counter, depending upon the instruction. The instruction descriptions use the notation (MP + index*2) to refer to the halfword in the message addressed by this index field. The index field and 16-bit displacements in the instruction stream shall be interpreted as unsigned quantities unless otherwise specified. The notation CHPC is used to refer to the program counter for the executing



channel program.

The opcode (the value of bits 0:7 in Hex) and function of each instruction is described below. Fetch of any instruction other than those described below shall cause an IOC error interrupt with fault code **Illegal.Opcode**.

13.9.1. Transfer Instructions. The transfer instructions read or write a block of data to or from the channel. These instructions have the form shown below:



The size field is used to specify the units of information to be transmitted on the channel. Bytes, halfwords, or words may be specified. If the parallelism of the channel hardware is greater than the number of bits in the specified transfer unit, each unit transferred shall be zero extended. The size field is encoded as follows:

Field	Data Size
0	reserved (Shall cause **Transfer.Unit** fault)
1	Byte (8-bits)
2	Halfword (16-bits)
3	Word (32-bits)

The information to be transferred is specified in general by a buffer address and unit count. This unit count is expressed in the number of bytes to be transferred. On read operations, the number of bytes actually transferred is made available as specified by the individual instructions. In the event a transfer is terminated prior to completion of the specified number of data transfers, the cause of the termination shall be indicated in the channel program status. Refer to section 13.9.2.

13.9.1.1. Indirect Buffer Specification. If the most significant bit of the buffer address specified in a transfer instruction is set, indirect buffer specification shall be assumed. Bits 1:31 of the buffer address specify the physical address of a transfer list. The format of such a list is shown in figure 13-1. The list contains 2 word entries describing segments of memory to be used in the transfer. The first word of each entry contains the initial transfer address; the second word contains the number of bytes in the memory segment. The specified transfer shall proceed using each memory segment in order, until the specified unit count is exhausted or the instruction terminates by some other means.

The indirect buffer list is assumed to be aligned on a word boundary. If bits 30:31 of the specified address are non-zero, a **List.Alignment** fault shall be generated.

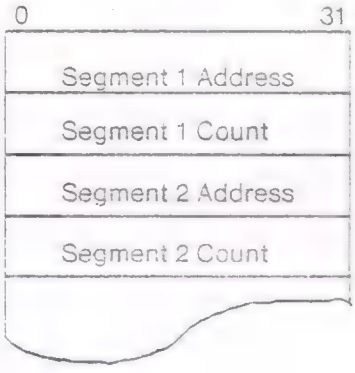


Figure 13-1: Indirect Buffer List

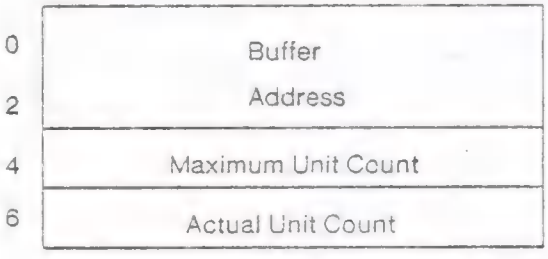
Read

READ

05:07
04 is reserved

Description:

A read operation is initiated on the channel. The data is read from the channel into the data buffer. The data buffer is specified by an operand block in the message. Bits 8:15 of the instruction specify the address of the operand block as an unsigned displacement (in halfwords) from the message pointer contents. The format of the data block is shown below. The actual number of units transferred is stored in the actual unit count halfword when the transfer terminates.



Read with Count

RDWCNT

09:0B
08 is reserved

Description:

RDWCNT functions similarly to READ except that the first unit transferred from the channel shall be interpreted as the actual count of bytes to follow. The data buffer is specified by an operand block in the message. Bits 8:15 of the instruction specify the address of the operand block as an unsigned displacement (in halfwords) from the message pointer contents. The format of the data block is the same as that for READ. The actual transfer count shall be stored as for READ

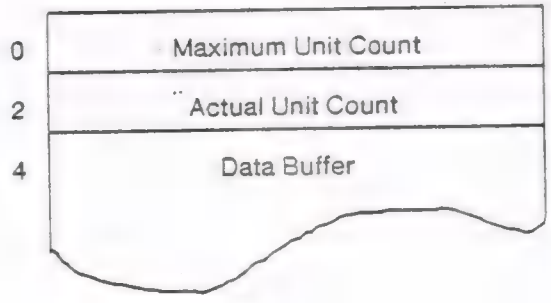
Read to Message

RDTMSG

0D:0F
0C is reserved

Description:

A read operation is initiated on the channel. The data is read into a buffer in the message. The transfer count is specified by an operand block in the message. Bits 8:15 of the instruction specify the address of the operand block as an unsigned displacement (in halfwords) from the message pointer contents. The format of the data block is shown below.



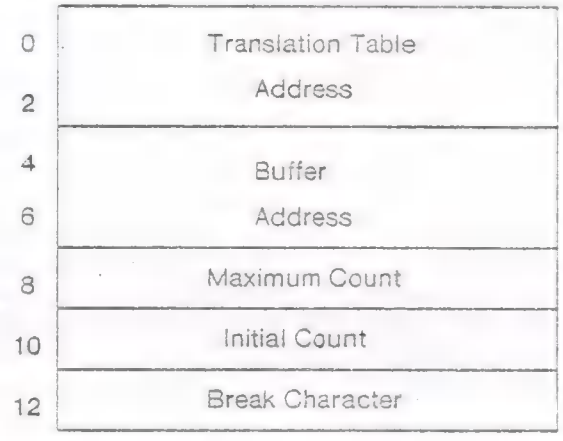
Read and Edit

RDEDIT

11
10, 12 and 13 are reserved

Description:

This instruction is restricted to byte transfers only. A translation table address and data buffer are specified by an operand block in the message. Bits 8:15 of the instruction specify the address of the operand block as an unsigned displacement (in halfwords) from the message pointer contents. The format of the data block is shown below.

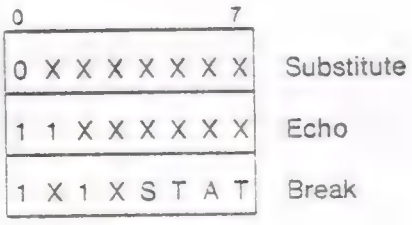


The maximum count halfword shall specify the maximum number of bytes to be placed in the buffer specified by the buffer address. When the transfer is initiated, the initial count halfword shall specify the number of bytes already stored in the buffer. Additional data shall be added to the buffer beginning at the next byte after those already in the buffer. On termination of the read, the initial count halfword shall be updated to reflect the number of bytes in the buffer.

When a byte is received from the channel, it shall be placed in the buffer. The value of the byte shall be added to the translation table address specified to determine the address of a control byte. The control byte shall specify one or more of the following additional actions:

Echo	The character read is written to the channel.
Substitute	The control byte is written to the channel.
Break	The transfer is terminated. The character read is stored in the break character halfword bits 0:7. Bits 8:15 shall be clear. Bits 4:7 of the control byte shall be placed in the channel program status bits 12:15.

The encoding of the control byte is shown below:



X = Don't Care

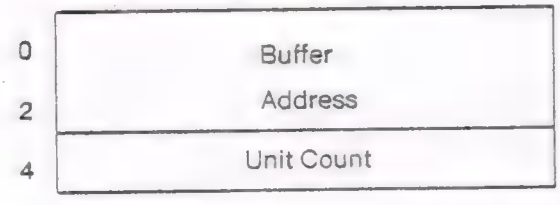
Write

WRITE

15:17
14 is reserved

Description:

A write operation is initiated on the channel. The data buffer is specified by an operand block in the message. Bits 8:15 of the instruction specify the address of the operand block as an unsigned displacement (in halfwords) from the message pointer contents. The format of the data block is shown below.



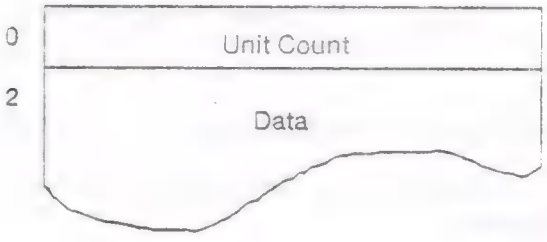
Write from Message

WRFMSG

19:18
18 is reserved

Description:

A write operation is initiated on the channel. The data is contained in the message. Bits 8:15 of the instruction specify the address of the data as an unsigned displacement (in halfwords) from the message pointer contents. The format of the data is shown below.



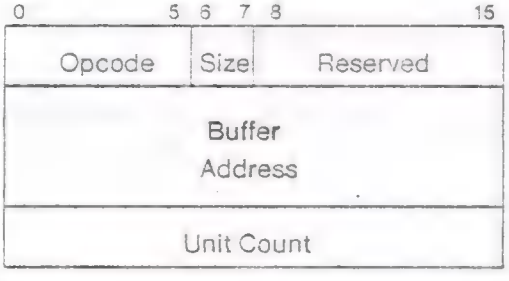
Write Literal

WRLIT

1D:1F, 32-bit address, 16-bit count
1C is reserved

Description:

A write operation is initiated on the channel. The address and unit count of the data are specified as literals in the instruction stream. The format of the WRLIT instruction is shown below.



13.9.2. Channel Status Instructions.

An IOC program has access to two types of status: channel program status and channel status. The channel program status is a 16 bit quantity that contains information about the last transfer operation requested by the channel program. The format of this halfword is shown in figure 13-2. The bits of the channel status halfword indicate which of several exceptional conditions may have caused termination of the transfer. The assigned fields are:

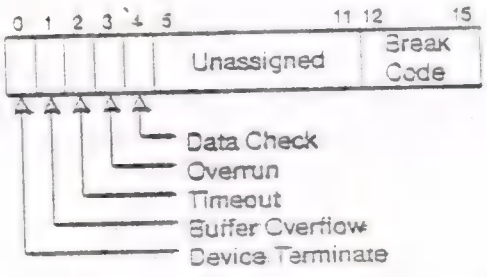


Figure 13-2: Channel Program Status

Device Termination The device signaled transfer completion prior to exhaustion of the transfer count.

Buffer Overflow On a read operation the length of the data transfer exceeded the maximum transfer count.

Timeout A transfer was not acknowleged within a channel dependent time.

Data Overrun Data was lost on a read due to the inability of the ICC to complete the transfer prior to the arrival of the next data.

Data Check An error was detected in the data received.

Break Code A program defined code generated by RDEDIT transfers.

The channel status is a 16 bit channel dependent quantity reflecting the control signals presented by the channel.

Read Channel Program Status

CPSTAT (MP + Index*2) <- channel program status 20

Description:

The channel program status is placed in the message at the specified location.

Read Channel Status

CSTAT	(MP + Index*2) <- channel status	21

Description:
The channel status is placed in the message at the specified location.

13.9.3. Control Instructions. The following instructions perform miscellaneous control functions.

Command

CMD	22
CMDL	23, 16-bit literal

Description:
A command is sent on the channel. The command is either a literal or the indexed halfword in the message, depending on the opcode. Depending on the channel, this instruction may assert control signals (as for an RS-449 interface) or cause a transfer of a command halfword (1553B or Parallel interface).

Load Message Pointer

LMP	MP <- (MP + Index*2)	24
LMPL	MP <- Literal	25, 32-bit Literal

Description:
The current message pointer is replaced by the indexed word in the message or by the 32-bit literal following the instruction. If the value loaded into the message pointer is zero, the channel program shall halt. LMP can be used to chain messages.

Indexed Load Message Pointer

XLMP	MP <- (MP + displ*2 + (MP + index*2)*4)	26, 16 bit displacement

Description:
XLMP allows a new message pointer to be loaded from an array of 32 bit vectors. The entry loaded is determined by a 16 bit selector in the message. Bits 8:15 of the instruction specify the index of the selector halfword. The displacement following the instruction specifies the location of the array. Twice this value plus MP forms the address of the zeroth entry in the array.

Add to Message Pointer

AMP MP <- MP + index*2 27

Description:
The signed index field of the instruction (bits 8:15) is added to the current contents of the message pointer register.

Branch

BRIO CHPC <- CHPC + index*2 28

Description:
Bits 8:15 of the instruction are interpreted as a signed halfword displacement from the next instruction address. The displacement is added to the incremented channel program counter.

Jump

JUMPIO CHPC <- literal 29, 32-bit literal

Description:
The literal is loaded into the Channel Program Counter.

Store Literal

STOLIT (MP + index*2) <- literal 2A, 16-bit literal

Description:
The indexed halfword in the message is replaced by the literal.

Add Literal

ADDLIT (MP + Index*2) <- (MP + index*2) + literal 2B, 16-bit literal

Description:
The literal following the instruction is added to the indexed halfword in the message.

Move within Message

MOVMSG (MP + Displ*2) <- (MP + index*2) 2C, 16-bit displ

Description:
The indexed halfword is copied into another position in the message. The position is specified by the halfword displacement following the instruction.

Masked Test and Branch

MTEST 2E, 16-bit mask displ, 16-bit branch displ
MTESTL 2F, 16-bit literal mask, 16-bit branch displ

Description:
The logical AND of the indexed halfword and the mask is performed. In MTEST, the mask is a halfword in the message at the address MP + mask displ*2. In MTESTL, the mask is a literal following the instruction. The branch displacement is interpreted as a signed halfword displacement from the incremented program counter. If the result is non-zero, the branch displacement times 2 is added to the address of the halfword following the branch displacement to form the next instruction address.

Case

CASE10 30, 16-bit mask, 16-bit shift cnt, {list of displacements}

Description:
The indexed halfword in the message is used as a selector for the case. The first literal following the instruction is the mask. The second literal following the instruction is the shift count. The logical AND of the selector and the mask is shifted right by the shift count. The resulting unsigned integer is used to select the displacement from the list of 16-bit literals that follow the instruction. The selected signed displacement times 2 is added to the address of the list to form the next instruction address.

Bit Case

BCASE 31, 16-bit count, {list of 16-bit displacements}

Description:
The 16-bit literal following the instruction contains the number of 16-bit displacements in the following list. The bit position number of the first set bit from the left in the indexed halfword is determined. If this position number is less than the list count then the position number selects one of the 16-bit displacements following the instruction. The selected displacement times 2 is added to the address of the list to form the next instruction address. Otherwise the CHPC is loaded with the address of the first halfword following the list (no branch is taken). If no bits were set in the indexed halfword no branch is taken.

Interrupt

INT	32, 16-bit interrupt code
INTL	33, 16-bit interrupt code

Description:

A processor interrupt is generated. For INT, the priority of this interrupt shall be determined by bits 11:15 of the indexed halfword. For INTL, the priority shall be determined by bits 11:15 of the instruction. The 16 bit literal following either instruction specifies the interruption code parameter of the interrupt.

Halt

HALT	2D

Description:

The channel program is halted.

14. Timer Support

The MCF architecture provides two types of timer support: time of day timer and an interval timers. These timers are accessible as registers in the I/O address space. Access to these registers is controlled by the memory management system.

14.1. Time of Day. There shall be a 32 bit register called the time of day clock. This register shall be a counter that is incremented every 10 ms. The contents of this register can be read or written by the software.

14.2. Interval Timers. There shall be four 32 bit registers called interval timers. These registers shall be counters that are decremented on the occurrence of particular events. These registers may be read or written by the software. Associated with each interval timer is a one byte field of the 32 bit Timer Control Register. The format of this register is shown in figure 14-1. Each byte of this register contains two fields: a mode field and a priority field. The mode field shall determine which events shall cause the corresponding interval timer to be decremented. The defined values of the mode field are:

000 Disabled. The corresponding interval timer shall not be decremented.

001 Time Base. The corresponding interval timer shall be decremented every 1 microsecond.

010 Instruction. The corresponding interval timer shall be decremented by each execution of an instruction while bit 0 (Kernel/Task) of the PSW is set. The timer shall also be decremented on each check for pending interrupts on interruptable vector instructions (such as MOVBK) while bit 0 of the PSW is set.

All other values of the mode field are reserved. When an interval timer is decremented to zero an interrupt shall be requested. The priority of this request shall be determined by the priority field in the Timer Control Register corresponding to the interval timer. The entry address of the procedure to be invoked shall be determined by the contents of the interval timer's interrupt vector. See section 15. This interrupt shall have no parameters.

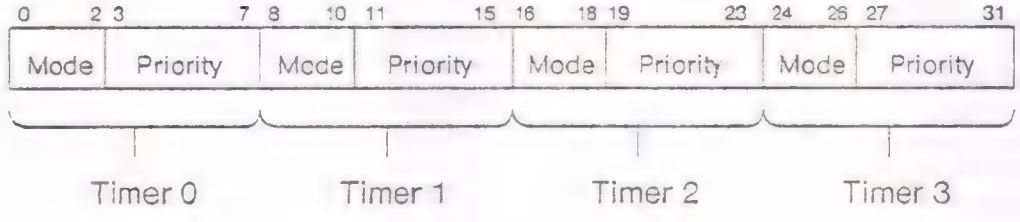


Figure 14-1: Timer Control Register

15. Assigned Physical Addresses

15.1. Interrupt Vector Assignments. The following table gives a list of interrupt and trap vectors defined by the architecture.

VECTOR NAME	ADDRESS (in hex)
Software interrupt	04
Parameterized interrupt	08
Memory management errors	0C
Memory system errors (Soft)	10
Memory system errors (Hard)	14
Power failure	18
Power restore	1C
Kernel save area pointer	20
Supervisor exception handler	24
Privileged instruction trap	28
Reserved	2C
Timer 0 interrupt	30
Timer 1 interrupt	34
Timer 2 interrupt	38
Timer 3 interrupt	3C

15.2. I/O Space Assignments. The top 2K bytes in the I/O space is dedicated to processor control registers. The following table lists the registers assigned in this page.

REGISTER NAME	ADDRESS (in hex)
Kernel context pointer	FFFFF800
Task context pointer	FFFFF804
Software interrupt request	FFFFF808
PSW	FFFFF80C
OPEX limit	FFFFF810
OPEX table address	FFFFF814
SVC limit	FFFFF818
SVC table address	FFFFF81C
Auxillary Status	FFFFF820
User map pointer	FFFFF824
Supervisor map pointer	FFFFF828
Timer control	FFFFF82C
Interval timer 0	FFFFF830
Interval timer 1	FFFFF834
Interval timer 2	FFFFF838
Interval timer 3	FFFFF83C
Time of day clock	FFFFF840
Reserved	FFFFF844 to FFFFFFFF

16. Integer Arithmetic

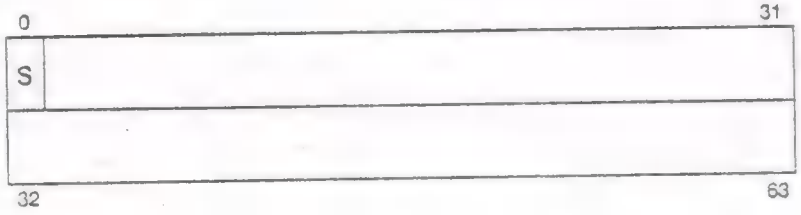
16.1. Integer Data Types. The MCF instruction set architecture supports integer data types of 8-bit, 16-bit, 32-bit and 64-bit sizes. These are termed byte, halfword, word and doubleword integers, respectively. Integers can be treated as either signed or unsigned. Signed integers are represented in 2's complement form. When *n* bits are used in the representation, signed integers range from -2^{n-1} to $2^{n-1}-1$ and unsigned integers range from 0 to $2^{n}-1$.

A **byte** is 8 contiguous bits starting on a byte boundary. The bits of decreasing significance are numbered from 0 to 7. A byte is specified by its address Addr. When the byte is interpreted as a signed integer, the most significant bit (bit 0) is the sign bit. The value of the signed integer is in the range -128 through 127.

A **halfword** is 2 contiguous bytes starting on a byte boundary. The bits of decreasing significance are numbered from 0 to 15. A halfword is specified by its address Addr, the address of the byte containing bit 0. When the halfword is interpreted as a signed integer, the most significant bit (bit 0) is the sign bit. The value of the signed integer is in the range -32768 through 32767.

A **word** is 4 contiguous bytes starting on a byte boundary. The bits of decreasing significance are numbered from 0 to 31. A word is specified by its address Addr, the address of the byte containing bit 0. When the word is interpreted as a signed integer, the most significant bit (bit 0) is the sign bit. The value of the signed integer is in the range -2^{31} through $2^{31}-1$.

A **doubleword** is 8 contiguous bytes starting on a byte boundary. The bits of decreasing significance are numbered from 63 to 0. A doubleword is specified by its address Addr, the address of the byte containing bit 0. When the doubleword is interpreted as a signed integer, the most significant bit (bit 0) is the sign bit. The value of the signed integer is in the range -2^{63} through $2^{63}-1$. The doubleword integer data type is only partly supported by the MCF instruction set architecture.



16.2. **Integer Arithmetic Instructions.** All operands involved in signed integer arithmetic instructions shall be sign extended internally to include enough bits so that all operations can be performed without any loss of information. If the instruction specifies a destination with fewer bits than the internal operation, the least significant bits of the result are stored in the destination. The truncate condition code (T) shall be set if the result cannot be correctly represented (as a signed integer) in the specified destination operand. The negative condition code (N) and the zero condition code (Z) are set to reflect the value of the destination operand except where otherwise noted. The carry condition code (C) is set by add and subtract instructions, and signifies carry from most significant (i.e. sign) bit of the internal result. If EAE is set (see section 6.14) and T is set at the completion of an instruction, a **Truncation** exception shall be raised.

All operands involved in unsigned integer arithmetic instructions shall be zero extended internally to 32-bits. If the instruction specifies a destination with fewer bits than the internal operation, the least significant bits of the result are stored in the destination. The truncate condition code (T) shall be set if the result cannot be correctly represented (as an unsigned integer) in the specified destination operand. The N and Z condition codes are set by treating the destination operand as a signed integer. The C condition code is set by a carry from the 32-bit internal result. If EAE is set and T is set at the completion of the instruction, a truncation exception shall be raised.

Integer Addition

ADD	R <- A + B	30, A, B, R
	R <- R + A	31, A, R

Operand types:
A, B, R 8-bit, 16-bit, 32-bit signed integers

Condition codes:
Z <- R Eql 0
N <- R Lss 0
T <- Set if the result is not representable in R, otherwise cleared
C <- Carry generated during 32-bit internal addition

Program exceptions:
Truncation

Description:
In the 3 operand format, the first operand is added to the second operand and the third operand is replaced by the sum. In the 2 operand format, the first operand is added to the second operand and the second operand is replaced by the sum.

Integer Subtraction

SUB	R <- B + not(A) + 1	32, A, B, R
	R <- R + not(A) + 1	33, A, R

Operand types:
A, B, R 8-bit, 16-bit, 32-bit signed integers

Condition codes:
Z <- R Eql 0
N <- R Lss 0
T <- Set if the result is not representable in R, otherwise cleared
C <- Carry generated during 32-bit internal addition

Program exceptions:
Truncation

Description:
In the 3 operand format, the 2's complement of the first operand is added to the second operand and the third operand is replaced by the sum. In the 2 operand format, the 2's complement of the first operand is added to the second operand and the second operand is replaced by the sum.

Integer Multiplication

MUL	R <- A x B	34, A, B, R
	R <- R x A	35, A, R

Operand types:
A, B, R 8-bit, 16-bit, 32-bit signed integers

Condition codes:
Z <- R Eql 0
N <- R Lss 0
T <- Set if the result is not representable in R, otherwise cleared
C Unaffected

Program exceptions:
Truncation

Description:
In the 3 operand format, the first operand is multiplied by the second operand and the third operand is replaced by the product. In the 2 operand format, the first operand is multiplied by the second operand and the second operand is replaced by the product.

Integer Division

DIV	R <- B / A	36, A, B, R
	R <- R / A	37, A, R

Operand types:
A, B, R 8-bit, 16-bit, 32-bit signed integers

Condition codes:
Z <- R Eql 0
N <- R Lss 0
T <- Set if the result is not representable in R, otherwise cleared
C Unaffected

Program exceptions:
Truncation
Illegal.Divisor

Description:
In the 3 operand format, the second operand is divided by the first operand and the third operand is replaced by the quotient. In the 2 operand format, the second operand is divided by the first operand and the second operand is replaced by the quotient. An Illegal.Divisor exception shall be raised if A = 0. When this exception occurs the operands are unaffected.

Integer Negate

NEG	R <- (-A)	3A, A, R
	R <- (-R)	3B, R

Operand types:

A, R 8-bit, 16-bit, 32-bit signed integers

Condition codes:

Z <- R Eql 0
N <- R Lss 0
T <- Set if the result is not representable in R, otherwise cleared
C Unaffected

Program exceptions:

Truncation

Description:

In the 2 operand format, the second operand is replaced by the negative value (i.e. 2's complement) of the first operand. In the 1 operand format, the operand is replaced by its negative value.

Integer Modulus

MOD R <- B MOD A 38, A, B, R

Operand types:

A, B, R 8-bit,16-bit,32-bit signed integers

Condition codes:

Z <- R Eql 0
N <- R Lss 0
T <- Set if the result is not representable in R, otherwise cleared
C Unaffected

Program exceptions:

Truncation

Description:

In MOD the third operand is replaced by the modulus of B and A. When A eql 0, the modulus is B. When A neq 0, the modulus is calculated as follows: the second operand is divided by the first operand, then the greatest integer less than or equal the quotient obtained is multiplied by the first operand, and finally the product is subtracted from the second operand.

```
B MOD A = B - A * floor(B/A), if A neq 0

B MOD 0 = B
```

Remainder

REM R <- remainder(B / A) 39, A, B, R

Operand types:
A, B, R 8-bit,16-bit,32-bit signed integers

Condition codes:
Z <- R Eql 0
N <- R Lss 0
T <- Set if the result is not representable in R, otherwise cleared
C Unaffected

Program exceptions:
Truncation
Illegal.Divisor

Description:
In REM the second operand is divided by the first operand and the third operand is replaced by the remainder. An **Illegal.Divisor** exception shall be raised if A = 0. When this exception occurs the operands are unaffected.

Extended Integer Multiplication

EMUL R <- A x B 4E, A, B, R

Operand types:
A, B 8-bit, 16-bit, 32-bit signed integers
R 16-bit, 32-bit, 64-bit signed integers

Condition codes:
Z <- R Eql 0
N <- R Lss 0
T <- Set if the result is not representable in R, otherwise cleared
C Unaffected

Program exceptions:
Truncation

Description:
The first operand is multiplied by the second operand and the third operand is replaced by the product. Since the third operand can be specified as a 64-bit signed integer,the product of 32-bit signed integers can be stored without any loss of information.

Extended Integer Divide

EDIV R2 <- B / A 4F, A, B, R1, R2
R1 <- remainder(B / A)

Operand types:

B, R2	16-bit, 32-bit, 64-bit signed integers
A, R1	8-bit, 16-bit, 32-bit signed integers

Condition codes:

Z <- R2 Eql 0
N <- R2 Lss 0
T <- Set if the result is not representable in R2, otherwise cleared
C Unaffected

Program exceptions:

Truncation
Illegal.Divisor

Description:

The second operand is divided by the first operand, and the third and fourth operands are replaced by the remainder and quotient, respectively. Since the second and fourth operands can be specified as 64-bit signed integers, extended precision integer division can be carried out and the result stored without any loss of information. An **Illegal.Divisor** exception shall be raised if A = 0. When this exception occurs the operands are unaffected.

Multiply Fixed Point

MULFIX 50, A, B, S, R

R <- (|A x B| shifted |S| places) x sign(A) x sign(B)

Operand types:

A, B, R	8-bit, 16-bit, 32-bit signed integers with implied radix point
S	8-bit, 16-bit, 32-bit signed integers

Condition codes:

Z <- R Eql 0
N <- R Lss 0
T <- Set if the result is not representable in R, otherwise cleared
C Unaffected

Program exceptions:

Truncation

Description:

The first operand is multiplied by the second operand and the absolute value of the product is first shifted either S places to the right, if S is positive, or S places to the left, if S is negative, and then the fourth operand is replaced by the result.

Since A, B, and R are fixed point they can be represented as $A \times 2^{-X}$, $B \times 2^{-Y}$, and $R \times 2^{-Z}$ where

X = number of bits to the right of the radix point in A
Y = number of bits to the right of the radix point in B
Z = number of bits to the right of the radix point in R

Therefore $R = A \times B \times 2^{Z-Y-X}$ meaning that in order to have R scaled properly the product must be shifted (Z-X-Y) places. It follows that S = Z - X - Y. However, we define S = X + Y - Z (i.e. -S = Z - Y - X) so that short literal form may be used for most cases of S.

Divide Fixed Point

DIVFIX 51, A, B, S, R

R <- ((|B| shifted |S| places) / |A|) x sign(A) x sign(B)

Operand types:

A, B, R	8-bit, 16-bit, 32-bit signed integers with implied radix point
S	8-bit, 16-bit, 32-bit signed integers

Condition codes:

Z <- R Eql 0
N <- R Lss 0
T <- Set if the result is not representable in R, otherwise cleared
C Unaffected

Program exceptions:

Truncation
Illegal.Divisor

Description:

The absolute value of the second operand is first shifted either S places to the right, if S is positive, or S places to the left, if S is negative, and then divided by the absolute value of the first operand. Finally, the fourth operand is replaced by the result.

Since A, B, and R are fixed point they can be represented as $A \times 2^{-X}$, $B \times 2^{-Y}$, and $R \times 2^{-Z}$ where

X = number of bits to the right of the radix point in A
Y = number of bits to the right of the radix point in B
Z = number of bits to the right of the radix point in R

Therefore $R = (B \times 2^{Z+X-Y}) / A$ meaning that in order to have R scaled properly B must be shifted (Z + X-Y) places. It follows that S = Z + X - Y.

Illegal.Divisor exception shall be raised if A = 0. When this exception occurs the operands are unaffected.

Increment

INC	R <- R + 1	55, R
INC2	R <- R + 2	52, R
INC4	R <- R + 4	53, R
INC8	R <- R + 8	54, R

Operand types:
R 8-bit, 16-bit, 32-bit signed integers

Condition codes:
Z <- R Eql 0
N <- R Lss 0
T <- Set if the result is not representable in R, otherwise cleared
C Unaffected

Program exceptions:
Truncation

Description:
In INC one is added to the operand and the operand is replaced by the sum. In INC2 two is added to the operand and the operand is replaced by the sum. In INC4 four is added to the operand and the operand is replaced by the sum. In INC8 eight is added to the operand and the operand is replaced by the sum.

Decrement

DEC	R <- R - 1	60, R

Operand types:
R 8-bit, 16-bit, 32-bit signed integers

Condition codes:
Z <- R Eql 0
N <- R Lss 0
T <- Set if the result is not representable in R, otherwise cleared
C Unaffected

Program exceptions:
Truncation

Description:
One is subtracted from the operand and the operand is replaced by the difference.

Add with Carry

ADDC R<- A + B + C 3E, A, B, R

Operand types:
A, B, R 8-bit, 16-bit, 32-bit signed integers

Condition codes:
Z <- R Eql 0
N <- R Lss 0
T <- Set if the result is not representable in R, otherwise cleared
C <- Carry generated during 32-bit internal addition

Program exceptions:
Truncation

Description:
The first and second operands and the contents of the condition code bit C are added, and the third operand is replaced by the sum.

Subtract with Carry

SUBC R<- B + not(A) + C 3F, A, B, R

Operand types:
A, B, R 8-bit, 16-bit, 32-bit signed integers

Condition codes:
Z <- R Eql 0
N <- R Lss 0
T <- Set if the result is not representable in R, otherwise cleared
C <- Carry generated during 32-bit internal addition

Program Exceptions:
Truncation

Description:
The 1's complement of the first operand, the second operand and the contents of the carry condition code (C) are added and the third operand is replaced by the sum.

Absolute Value

ABS R <- |A| 3C, A, R

Operand types:
A, R 8-bit, 16-bit, 32-bit signed integers

Condition codes:
Z <- R Eql 0
N <- R Lss 0
T <- Set if the result is not representable in R, otherwise cleared
C Unaffected

Program exceptions:
Truncation

Description:
The second operand is replaced by the absolute value of the first operand.

Unsigned Addition

ADDU R <- A + B 56, A, B, R

Operand types:
A, B, R 8-bit, 16-bit, 32-bit unsigned integers

Condition codes:
Z <- R Eql 0
N <- R Lss 0
T <- Set if the result is not representable in R, otherwise cleared
C <- Carry generated during 32-bit internal addition

Program exceptions:
Truncation

Description:
The operands are zero extended to 32-bits. The first operand is added to the second operand and the third operand is replaced by the sum.

Unsigned Subtraction

SUBU R <- B + not(A) + 1 57, A, B, R

Operand types:
A, B, R 8-bit, 16-bit, 32-bit unsigned integers

Condition codes:
Z <- R Eql 0
N <- R Lss 0
T <- Set if the result is not representable in R, otherwise cleared
C <- Carry generated during 32-bit internal addition

Program exceptions:
Truncation

Description:
The operands are zero extended to 32-bits. The 2's complement of the first operand is added to the second operand and the third operand is replaced by the sum.

Unsigned Multiplication

MULU R <- A x B 58, A, B, R

Operand types:
A, B, R 8-bit, 16-bit, 32-bit unsigned integers

Condition codes:
Z <- R Eql 0
N <- R Lss 0
T <- Set if the result is not representable in R, otherwise cleared
C Unaffected

Program exceptions:
Truncation

Description:
The source operands are zero extended to 32-bits. The first operand is multiplied by the second operand and the third operand is replaced by the product.

Unsigned Division

DIVU R <- B / A 59, A, B, R

Operand types:

A, B, R 8-bit, 16-bit, 32-bit unsigned integers

Condition codes:

Z <- R Eql 0
N <- R Lss 0
T <- Set if the result is not representable in R, otherwise cleared
C Unaffected

Program exceptions:

Truncation
Illegal.Divisor

Description:

The source operands are zero extended to 32-bits. The second operand is divided by the first operand and the third operand is replaced by the quotient. An Illegal.Divisor exception shall be raised if A = 0. When this exception occurs the operands are unaffected.

17. Logical Instructions

All operands involved in logical instructions shall be zero extended internally to include enough bits so that all operations can be performed without any loss of information. If the instruction specifies a destination with fewer bits than the internal operation, the least significant bits of the result are stored in the destination.

In general, the negative (N) and zero (Z) condition codes are set to reflect the value of the result stored (which is treated as a signed integer), except where otherwise noted. The truncate (T) condition code and the carry (C) condition code are unaffected, unless otherwise noted.

Logical Not

NOT	R <- not(A)	46, A, R
	R <- not(R)	47, R

Operand types:

A, R 8-bit, 16-bit, 32-bit logical

Condition codes:

Z <- R Eql 0
N <- R Lss 0
T Unaffected
C Unaffected

Program exceptions:

None

Description:

In the 2 operand format, the second operand is replaced by the logical complement of the first operand. In the 1 operand format, the operand is replaced by its logical complement. The condition code bits are set depending on the value of the operand stored which is treated as a signed integer.

Logical And

AND	R <- A and B	48, A, B, R
	R <- R and A	49, A, R

Operand types:
A, B, R 8-bit, 16-bit, 32-bit logical

Condition codes:
Z <- R Eql 0
N <- R Lss 0
T Unaffected
C Unaffected

Program exceptions:
None

Description:
In the 3 operand format, the logical AND function is performed on the first two operands, and the third operand is replaced by the result. In the 2 operand format, the logical AND function is performed on the first two operands, and the second operand is replaced by the result. The condition code bits are set depending on the value of the operand stored which is treated as a signed integer.

Logical Or

OR	R <- A or B	4A, A, B, R
	R <- R or A	4B, A, R

Operand types:
A, B, R 8-bit, 16-bit, 32-bit logical

Condition codes:
Z <- R Eql 0
N <- R Lss 0
T Unaffected
C Unaffected

Program exceptions:
None

Description:
In the 3 operand format, the logical OR function is performed on the first two operands, and the third operand is replaced by the result. In the 2 operand format, the logical OR function is performed on the first two operands, and the second operand is replaced by the result. The condition code bits are set depending on the value of the operand stored which is treated as a signed integer.

Logical Exclusive Or

XOR R <- A xor B 4C, A, B, R

Operand types:
A, B, R 8-bit, 16-bit, 32-bit logical

Condition codes:
Z <- R Eql 0
N <- R Lss 0
T Unaffected
C Unaffected

Program exceptions:
None

Description:
The logical Exclusive OR function is performed on the first two operands, and the third operand is replaced by the result. The condition code bits are set depending on the value of the operand stored, which is treated as a signed integer.

Count One Bits

COB 5F, A, R

R <- Number of bits set in A

Operand types:
A 8-bit, 16-bit, 32-bit logical
R 8-bit, 16-bit, 32-bit unsigned integers

Condition codes:
Z <- R Eql 0
N <- R Lss 0
T Unaffected
C Unaffected

Program exceptions:
None

Description:
In COB, the second operand is replaced by the value which represents the number of bits set in the first operand.

18. Shift and Rotate Instructions

Arithmetic Shift

ASH R <- B shifted |A| bit positions 5C, A, B, R

Operand types:

A, B, R 8-bit, 16-bit, 32-bit signed integers

Condition codes:

Z <- R Eql 0
N <- R Lss 0
T <- Set if the sign bit changes at any time during the shift operation,
otherwise cleared
C Unaffected

Program exceptions:

Truncation

Description:

In ASH, the second operand is arithmetically shifted by the number of bits specified by the first operand, and the third operand is replaced by the result. When the first operand is positive, a left shift is performed and 0's are inserted into the least significant bits. When the first operand is negative, a right shift is performed and the sign bit is replicated. The values of the first two operands are unaffected.

Rotate

ROT R <- B rotated |A| bit positions 5D, A, B, R

Operand types:

A	8-bit, 16-bit, 32-bit signed integers
B, R	8-bit, 16-bit, 32-bit logical

Condition codes:

Z <- R Eql 0
N <- R Lss 0
T Unaffected
C Unaffected

Program exceptions:

None

Description:

In ROT, the second operand is logically rotated by the number of bits specified by the first operand, and the third operand is replaced by the result. When the first operand is positive, a left rotate is performed.When the first operand is negative, a right rotate is performed. Rotation is performed on the length of the second operand (i.e. B). The values of the first two operands are unaffected.

The Z and N bits are set depending on the value of the result stored, which is treated as a signed integer.

Logical Shift

LSH R <- B shifted |A| bit positions 5E, A, B, R

Operand types:

A	8-bit, 16-bit, 32-bit signed integers
B, R	8-bit, 16-bit, 32-bit logical

Condition codes:

Z <- R Eql 0
N <- R Lss 0
T Unaffected
C Unaffected

Program exceptions:

None

Description:

In LSH, the second operand is shifted logically by the number of bits specified by the first operand, and the third operand is replaced by the result. When the first operand is positive, a left shift is performed and 0's are inserted into the least significant bits. When the first operand is negative, a right shift is performed and 0's are inserted into the most significant bits. The values of the first two operands are unaffected.

The Z and N bits are set depending on the value of the result stored, which is treated as a signed integer.

19. Move and Clear Instructions

Move Arithmetic (Sign Extended)

MOV R <- A 40, A, R

Operand types:
A, R 8-bit, 16-bit, 32-bit signed integer

Condition codes:
Z <- R Eql 0
N <- R Lss 0
T Set if information lost during truncation, otherwise cleared
C Unaffected

Program exceptions:
None

Description:
Operand A is copied (sign extended or truncated as necessary) into R.

Move Logical (Zero Extended)

MOVL R <- A 41, A, R

Operand types:
A, R 8-bit, 16-bit, 32-bit logical

Condition codes:
Z <- R Eql 0
N <- R Lss 0
T Unaffected
C Unaffected

Program exceptions:
None

Description:
The operand A is copied (Zero extended or truncated as necessary) into R.

Move Address

MOVA R <- A 61, A, R

Operand types:
A Address Operand
R 8-bit, 16-bit, 32-bit logical

Condition codes:
Z <- R Eql 0
N <- R Lss 0
T Unaffected
C Unaffected

Program exceptions:
None

Description:
The address operand A is copied into R. If R is smaller than 32-bits, the high order part of A is lost.

Clear

CLR R <- 0 45, R

Operand types:
R 8-Bit, 16-bit, 32-bit logical

Condition codes:
Z <- 0
N <- 0
T Unaffected
C Unaffected

Program exceptions:
None

Description:
The operand R is set to 0 (all bits cleared).

Exchange

EXCH	tmp <- A next A <- B next B <- tmp	3D, A, B

Operand types:
A, B 8-bit, 16-bit, 32-bit logical

Condition codes:
Z <- B Eql 0 (after transfer)
N <- B Lss 0 (after transfer)
T Unaffected
C Unaffected

Program exceptions:
None

Description:
The contents of the two operands A and B are exchanged. The condition codes are set to reflect the final transfer into B.

Move To Stack

MTS	S <- S - size(A) next (S) <- A	70, A, S

Operand types:
A, S 8-bit, 16-bit, 32-bit logical

Condition codes:
Unaffected

Program exceptions:
None

Description:
The operand A is copied onto the push down stack defined by S. Size(A) is the size of A in bytes. S is zero extended to form a 32-bit address.

Move From Stack

MFS	A <- (S) next S <- S + size(A)	72, S, A

Operand types:
A,S 8-bit, 16-bit, 32-bit logical

Condition codes:
Z <- A Eql 0
N <- A Lss 0
T Unaffected
C Unaffected

Program exceptions:
None

Description:
The top item on the stack defined by S is removed and placed in A. Size(A) is the size of A in bytes. S is zero extended to form a 32-bit address.

Push onto SP Stack

PUSH	SP <- SP - 4 next (SP) <- A	71, A

Operand types:
A 8-bit, 16-bit, 32-bit logical

Condition codes:
Unaffected

Program exceptions:
None

Description:
If operand A is less than 32-bits, it is zero extended to 32-bits. The 32-bit item is copied onto the push down stack defined by register 1 (the stack pointer SP).

Pop from SP Stack

POP	A <- (SP) next SP <- SP + 4	73, A

Operand types:

A 8-bit, 16-bit, 32-bit logical

Condition codes:

Z <- A Eql 0
N <- A Lss 0
T Unaffected
C Unaffected

Program exceptions:

None

Description:

The top 32-bit item is removed from the stack defined by register 1 (the stack pointer SP) and is placed in operand A. If A is less than 32-bits, only the low order part is stored and no indication is given.

20. Compare and Test Instructions

Compare (Sign Extended)

CMP 42, A, B

Operand types:
A, B 8-bit, 16-bit, 32-bit signed integers

Condition codes:
Z <- A Eql B
N <- A Lss B
T Unaffected
C Unaffected

Program exceptions:
None

Description:
Arithmetic comparison is carried out on signed integers A and B, and the condition code bits are set accordingly. Operands involved in comparison are unaffected.

Compare Unsigned

CMPU 43, A, B

Operand types:
A, B 8-bit, 16-bit, 32-bit unsigned integers

Condition codes:
Z <- A Eql B
N <- A Lssu B
T Unaffected
C Unaffected

Program exceptions:
None

Description:
A comparison is made between unsigned integers A and B, and the condition code bits are set accordingly. Operands involved in the comparison are unaffected.

Compare and Swap

CMPS 4D, A, B, C

If B eql C then B <- A else C <- B

Operand types:

A, B, C 8-bit, 16-bit, 32-bit logical

Condition codes:

Z <- B Eql C (before transfer)
N <- B Lssu C (before transfer)
T Unaffected
C Unaffected

Program exceptions:

None

Description:

The second and third operands are compared. If the result of the comparison is equality, the second operand is replaced by the first operand, otherwise the third operand is replaced by the second operand. This is an interlocked operation. A serialization function preceeds operand B fetch. No other memory access to B is permitted until completion of the store of operand B, or the negative result of the condition.

Compare within Bounds

CMPWB 5A, A, B1, B2

Operand types:
A, B1, B2 8-bit, 16-bit, 32-bit signed integers

Condition codes:
Z <- (A Geq B1) AND (A Leq B2)
N <- (A Lss B1)
T Unaffected
C Unaffected

Program Exceptions:
None

Description:
A check whether the value of the first operand falls within the bounds specified by the second (lower bound) and third (upper bound) operands is performed.

The N and Z condition code bits specify the result of the check as follows:

Z = 1; N = 0 - value is within bounds;
Z = 0; N = 1 - value is less than the lower bound;
Z = 0; N = 0 - value is greater than the upper bound;

The operands involved are unaffected.

Compare within Bounds and Take Exception (Range Check)

RANGE 5B, A, B1, B2

If (A Lss B1) OR (A Gtr B2) then Range.Error Exception

Operand types:
A, B1, B2 8-bit, 16-bit, 32-bit signed integers

Condition codes:
Unaffected

Program exceptions:
Range.Error

Description:
A check whether the value of the first operand falls within the range specified by the second (lower bound) and third (upper bound) operands is performed, and the **Range.error** exception is raised if it is out of range. The operands involved are unaffected.

Test Integer

TEST 44, A

Operand types:
A 8-bit, 16-bit, 32-bit signed integers

Condition codes:
Z <- A Eql 0
N <- A Lss 0
T Unaffected
C Unaffected

Program exceptions:
None

Description:
Arithmetic comparison is carried out on the operand (signed integer) and zero, and the condition code bits are set accordingly. Operands involved are unaffected.

Set Based on Condition

EQL	A <- sign.extended (Z)	24, A
NEQ	A <- sign.extended not(Z)	26, A
LSS	A <- sign.extended (N)	28, A
GTR	A <- sign.extended not(N or Z)	2E, A
LEQ	A <- sign.extended (N or Z)	2C, A
GEQ	A <- sign.extended not(N)	2A, A

Operand types:

A 8-bit, 16-bit, 32-bit logical

Condition codes:

Z <- A Eql 0
N <- A Lss 0
T Unaffected
C Unaffected

Program Exceptions:

None

Description:

The operand is set (all 1's) if the condition is true. These instructions are intended to follow compare instructions which set the condition code bits.

21. Control Instructions

Control instructions use displacements to change the execution control. Displacements are 8-bit or 16-bit signed integers that follow the instruction in the code stream. The displacement is fetched, sign extended to 32-bits and added to the program counter (PC). The PC used in the displacement calculations of branch instructions is the address of the displacement in the code stream.

Jump

JUMP PC <- A 63, A

Operand types:
A Address Operand

Condition codes:
Unaffected

Program exceptions:
None

Description:
The PC is replaced by the Address operand.

Branch

BR 0E, 8-bit-displacement
0F, 16-bit-displacement

PC <- PC + displacement

Operand types:
None

Condition codes:
Unaffected

Program exceptions:
None

Description:
The sign-extended branch displacement is added to the PC and the PC is replaced by the result.

Branch on Equal

BEQL

14, 8-bit-displacement
15, 16-bit-displacement

If Z then PC <- PC + displacement

Operand types:
None

Condition codes:
Unaffected

Program exceptions:
None

Description:
If the Z bit is set then the PC is replaced with the sum of the PC and the sign-extended branch displacement. If the branch condition is not met then the PC is set to point to the next instruction in the instruction stream.

Branch on Not Equal

BNEQ

16, 8-bit-displacement
17, 16-bit-displacement

If not Z then PC <- PC + displacement

Operand types:
None

Condition codes:
Unaffected

Program exceptions:
None

Description:
If the Z bit is reset then the PC is replaced with the sum of the PC and the sign-extended branch displacement. If the branch condition is not met then the PC is set to point to the next instruction in the instruction stream.

Branch on Less than or Equal

BLEQ

1C, 8-bit-displacement
1D, 16-bit-displacement

If Z or N then PC <- PC + displacement

Operand types:
None

Condition codes:
Unaffected

Program exceptions:
None

Description:
If the Z bit is set or the N bit is set then the PC is replaced with the sum of the PC and the sign-extended branch displacement. If the branch condition is not met then the PC is set to point to the next instruction in the instruction stream.

Branch on Less than

BLSS

18, 8-bit-displacement
19, 16-bit-displacement

If N then PC <- PC + displacement

Operand types:
None

Condition codes:
Unaffected

Program exceptions:
None

Description:
If the N bit is set then the PC is replaced with the sum of the PC and the sign-extended branch displacement. If the branch condition is not met then the PC is set to point to the next instruction in the instruction stream.

Branch on Greater than or Equal

BGEQ

1A, 8-bit-displacement
1B, 16-bit-displacement

If Z or (not N) then PC <- PC + displacement

Operand types:
None

Condition codes:
Unaffected

Program exceptions:
None

Description:
If the Z bit is set or the N bit is reset then the PC is replaced with the sum of the PC and the sign-extended branch displacement. If the the branch condition is not met the PC is set to point to the next instruction in the instruction stream.

Branch on Greater than

BGTR

1E, 8-bit-displacement
1F, 16-bit-displacement

If (not N) and (not Z) then PC <- PC + displacement

Operand types:
None

Condition codes:
Unaffected

Program exceptions:
None

Description:
If the Z bit is reset and the N bit is reset then the PC is replaced with the sum of the PC and the sign-extended branch displacement. If the branch condition is not met then the PC is set to point to the next instruction in the instruction stream.

Branch on Carry Set

BCS

10, 8-bit-displacement
11, 16-bit-displacement

If C then PC <- PC + displacement

Operand types:
None

Condition codes:
Unaffected

Program exceptions:
None

Description:
If the C bit is set then the PC is replaced with the sum of the PC and the sign-extended branch displacement. If the branch condition is not met then the PC is set to point to the next instruction in the instruction stream.

Branch on Carry Clear

BCC

12, 8-bit-displacement
13, 16-bit-displacement

If not C then PC <- PC + displacement

Operand types:
None

Condition codes:
Unaffected

Program exceptions:
None

Description:
If the C bit is reset then the PC is replaced with the sum of the PC and the sign-extended branch displacement. If the branch condition is not met then the PC is set to point to the next instruction in the instruction stream.

Branch on Truncate Set

BTS

20, 8-bit-displacement
21, 16-bit-displacement

If T then PC <- PC + displacement

Operand types:
None

Condition codes:
Unaffected

Program exceptions:
None

Description:
If the T bit is set then the PC is replaced with the sum of the PC and the sign-extended branch displacement. If the branch condition is not met then the PC is set to point to the next instruction in the instruction stream.

Branch on Truncate Clear

BTC

22, 8-bit-displacement
23, 16-bit-displacement

If not T then PC <- PC + displacement

Operand types:
None

Condition codes:
Unaffected

Program exceptions:
None

Description:
If the T bit is reset then the PC is replaced with the sum of the PC and the sign-extended branch displacement. If the branch condition is not met then the PC is set to point to the next instruction in the instruction stream.

Case

CASE 7C, Sel, Base, Num, 16-bit-Displ[0], ... , 16-bit-Displ[Num-1]

If (Sel - Base) Lssu Num Then PC <- PC + displacement[Sel - Base]
Otherwise PC <- PC + (2 x Num)

Operand types:
Sel, Base, Num — 8-bit, 16-bit, 32-bit signed integer

Condition codes:
Unaffected

Program exceptions:
None

Description:
The selector operand (Sel) is subtracted from the base operand (Base). This value is used in an unsigned comparison with the number of cases Num. If the value is less than Num then the PC is replaced by the sum of the PC and a branch displacement selected by the value Sel - Base. Otherwise the PC is set to point to the byte after the last displacement.

After operand evaluation the PC is pointing at displacement[0], not the next instruction. Therefore the branch displacements are relative to the address of displacement[0].

Loop

LOOP 27, Inc, Ctr, Lim, 16-bit-displacement

```
Ctr <- Ctr + Inc next
If (Inc geq 0 and Ctr leq Lim) or
  (Inc lss 0 and Ctr geq Lim)
  Then PC <- PC + displacement
```

Operand types:
Inc, Ctr, Lim 8-bit, 16-bit, 32-bit signed integer

Condition codes:
Unaffected

Program exceptions:
None

Description:
The signed value of Inc is added to Ctr and Ctr is updated. If one of the comparisons with the limit (Lim) is within range, the signed displacement is added to the PC. Otherwise the PC is set to point to the next instruction.

Increment and Branch on Less than or Equal to

IBLEQ 2D, Ctr, Lim, 16-bit-displacement

```
Ctr <- Ctr + 1 next
If Ctr leq Lim Then PC <- PC + displacement
```

Cperand types:
Ctr, Lim 8-bit, 16-bit, 32-bit signed integer

Condition codes:
Unaffected

Program exceptions:
None

Description:
One is added to the counter (Ctr). If the new value of the counter is less than or equal to the limit (Lim), the signed displacement is added to the program counter. Otherwise the PC is set to point to the next instruction.

Increment and Branch on Less than

IBLSS 29, Ctr, Lim, 16-bit-displacement

Ctr <- Ctr + 1 next
If Ctr lss Lim Then PC <- PC + displacement

Operand types:
Ctr, Lim 8-bit, 16-bit, 32-bit signed integer

Condition codes:
Unaffected

Program exceptions:
None

Description:
One is added to the counter (Ctr). If the new value of the counter is less than the limit (Lim), the signed displacement is added to the program counter. Otherwise the program counter is set to point to the next instruction.

Decrement and Branch on Greater than or Equal to

DBGEQ 2B, Ctr, Lim, 16-bit-displacement

Ctr <- Ctr - 1 next
If Ctr geq Lim Then PC <- PC + displacement

Operand types:
Ctr, Lim 8-bit, 16-bit, 32-bit signed integer

Condition codes:
Unaffected

Program exceptions:
None

Description:
One is subtracted from the counter (Ctr). If the new value of the counter is greater than or equal to the limit (Lim), then the signed displacement is added to the program counter. Otherwise, the program counter is set to point to the next instruction.

Decrement and Branch on Greater

DBGTR 2F, Ctr, Lim, 16-bit-displacement

Ctr <- Ctr - 1 next
If Ctr gtr Lim Then PC <- PC + displacement

Operand types:
Ctr, Lim 8-bit, 16-bit, 32-bit signed integer

Condition codes:
Unaffected

Program exceptions:
None

Description:
One is subtracted from the counter (ctr). If the new counter is greater than the limit (Lim), then the signed displacement is added to the program counter. Otherwise the program counter is set to point to the next instruction.

Enter Loop

ENTLP 25, Init, 16-bit displacement

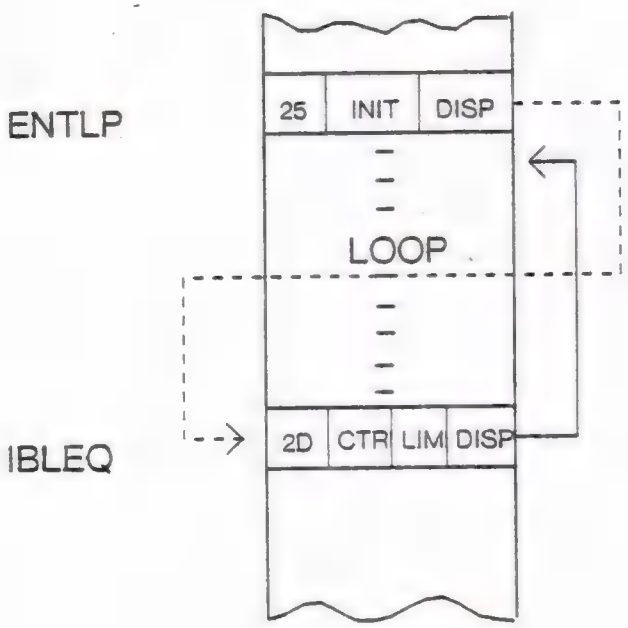
Operand types:
Init 8-bit, 16-bit, 32-bit signed integer

Condition codes:
Unaffected

Program exceptions:
Bad.Displacement

Description:
The 16-bit displacement, when added to the PC, will reference one of the loop control instructions (e.g. IBLEQ) in the instruction stream (otherwise a **Bad.Displacement** exception shall be generated). The contents of Init (the initial value) are placed in the counter (Ctr) operand specified by the loop control instruction, the check specified by the instruction is performed, and branching is based on the check.

ENTER LOOP INSTRUCTION



22. Procedure Call and Return Instructions

Call Procedure

CALL

64, Proc, Nparms, P1, ..., Pn
64, Proc, P1, ..., Pn

Operand types:

Proc	Address Operand
Nparms	8-bit, 16-bit, 32-bit unsigned integer
P1, ..., Pn	All types of operands allowed in Parameter list

Condition codes:

Unaffected

Program exceptions:

Invalid supervisor trap

Description:

The CALL instruction invokes a procedure with a parameter list. The first operand of the CALL instruction shall be interpreted as the address of the procedure entry to be invoked. If the most significant bit of this address is set (bit 0), and the current PSW has bit 17 (Supervisor) clear, a memory management trap shall occur. The reported error shall be **Invalid.Supervisor** (refer to section 12.4). Otherwise, the specified procedure entry shall be invoked as described in section 8.3. The remaining operands of the call instruction shall form the parameter list for this invocation. The privilege bit (bit 15) of the PSW shall remain unchanged.

Call Unprivileged Procedure

CALLU	65, Proc, Nparms, P1, ..., Pn 65, Proc, P1, ..., Pn

Operand types:

Proc	Address Operand
Nparms	8-bit, 16-bit, 32-bit unsigned integer
P1, ..., Pn	All types of operands allowed in Parameter list

Condition codes:

Unaffected

Program exceptions:

Invalid supervisor trap

Description:

The CALLU instruction operates in the same way as the CALL instruction Except the privilege bit in the PSW of the called context shall be cleared.

Return from Procedure

RET 67

Operand types:
None

Condition codes:
Loaded depending on BASE bit of PSW

Program exceptions:
None

Description:
The RET instruction returns from a procedure. The function of this instruction shall depend upon the BASE bit (bit 16) of the current PSW. If the base bit is clear the current procedure context has a caller; If the base bit is set the current context is the base of the execution context. Because of this distinction, the RET instruction shall function differently in these cases:

BASE = 0 — The current procedure context shall be removed from the context stack. The caller's context (which is now at the top of the stack) shall be restored. Bits 13:31 of the PSW shall be restored to the value specified in the caller's context. Bits 0:12 of the PSW shall be unchanged.

BASE = 1 — The current procedure context shall be removed from the context stack. The Last Mode bit of the current PSW (bit 1) shall select the context stack to become the current context stack. The procedure context at the top of this context stack shall be restored, including the full PSW contained in this context.

Call Supervisor

SVC		66, Index, Nparms, P1, ..., Pn 66, Index, P1, ..., Pn

Operand types:

Index	8-bit, 16-bit, 32-bit unsigned integer
P1, ..., Pn	All types of operands allowed in parameter list

Condition codes:

Unaffected

Program exceptions:

None

Description:

The supervisor call instruction provides a means of calling a supervisor specified procedure in a protected manner. The first operand of this instruction shall be evaluated as an unsigned integer. The result shall be used as an index for a vectored call operation using the SVC vector registers as described in section 8.5. The procedure entry determined by the vectoring operation shall be invoked using the current context stack. The remaining operands of the SVC shall be used as a parameter list for the call.

Jump to Subroutine

JSR	SP <- SP - 4 next (SP) <- PC next PC <- A	63, A

Operand types:

A	Address Operand

Condition codes:

Unaffected

Program exceptions:

None

Description:

The program counter is stored on the push down stack defined by register 1 (SP). The program counter is then set to A.

Return from Subroutine

RSR	PC <- (SP) next SP <- SP + 4	68

Operand types:
None

Condition codes:
Unaffected

Program exceptions:
None

Description:
The top item on the register 1 (SP) stack is removed and placed in the program counter.

23. Task Control Instructions

Load Task

LTASK 00, A

Operand types:
A Address Operand

Condition codes:
Based on PSW of loaded task context

Program exceptions:
Privileged Instruction Trap
Context.Alignment

Description:
The Load TASK instruction prepares a new task for execution by loading its memory map pointer and context pointer into the appropriate registers. The operand of this instruction shall specify the address of a two word block. The first word (lowest address) will contain the value of the context pointer. This value shall be placed in the Task Context Pointer register. The second word will contain the physical address of the task's memory map. This address shall be placed in the User Map Pointer register. The Context Pointer is required to be aligned on a word boundary. If the value specified for the context pointer has bits 30 or 31 set, a context.Alignment exception shall be raised.

This instruction shall have the implementation dependent effect of forcing any context stack or memory map caches to be consistent with the specified map and context stack in memory. If LTASK is executed by a procedure on the Task Context stack, the result is unpredictable.

Store Task

STASK 01, A

Operand types:
A Address Operand

Condition codes:
Unaffected

Program exceptions:
Privileged Instruction Trap

Description:
The Store TASK instruction is used to prepare a task for suspension by saving its context in memory. The operand A shall specify the address of a two word block with the same format as that used by LTASK. The Task Context Pointer shall be stored in the first word and the User Map Pointer shall be stored in the second word.

This instruction shall have the implementation dependent effect of forcing all cached Task context into memory. This insures that the task can be restarted by a subsequent LTASK.

Start Task

TSTART	02, A
PSTART	03, A

Operand types:

A 8-bit, 16-bit, 32-bit unsigned integer

Condition codes:

Unaffected

Program exceptions:

Privileged Instruction Trap
Specification.Error

Description:

The TSTART and PSTART instructions cause the processor to resume execution of the topmost procedure context on the context stack specified by the operand. If PSTART is executed, the top execution context shall be removed from the current context stack. This means that procedure contexts are removed until one with the BASE bit set in its PSW is discarded. If TSTART is executed, the current context shall simply be suspended. Next both TSTART and PSTART shall cause execution to resume with the topmost context on the specified context stack. The operand is decoded as follows:

Operand Value	Specification
0	Kernel Context Stack
1	Task Context Stack
Otherwise	Specification.Error

A Specification.Error shall be raised if the value of the operand is not zero or one.

Start Task Setting Exception

TRAISE	04, A, B
PRAISE	05, A, B

Operand types:

A, B 8-bit, 16-bit, 32-bit unsigned integer

Condition codes:

Unaffected

Program exceptions:

Privileged Instruction Trap
Specification.Error

Description:

The TRAISE and PRAISE instructions cause an exception to be raised in the topmost procedure on the context stack specified by the second operand. The exception code shall be specified by the first operand of the instruction. PRAISE shall cause the top execution context to be removed from the current context stack. TRAISE shall simply suspend execution of the current context. Next both TRAISE and PRAISE shall cause an exception to be raised in the topmost procedure context on the specified context stack. Execution shall begin with the processing of this exception. The second operand is decoded as follows:

Operand Value	Specification
0	Kernel Context Stack
1	Task Context Stack
Otherwise	Specification.Error

A **Specification.Error** shall be raised if the value of the operand is not zero or one.

Initiate Task

TINIT	06, A, B
PINIT	07, A, B

Operand types:

A	Address Operand
B	8-bits, 16-bits, 32-bits logical

Condition codes:

Unaffected

Program exceptions:

Privileged Instruction Trap

Description:

The TINIT and PINIT instructions invoke a task. Specifically, they create an execution context on the specified context stack and begin its execution. The first operand shall specify the entry address of the procedure to be entered as the "main program". The second operand shall specify bits 0:15 of the PSW for the new execution context. Note that bit zero selects the context stack on which the execution context will be built. PINIT shall remove the top execution context from the current context stack. TINIT shall simply suspend execution of the current context. Both TINIT and PINIT shall then use the specified entry address and PSW to invoke the procedure as described in section 8.3.

24. Exception Handling Instructions

Raise Exception

RAISE 69, A

Operand types:
A 8-bit, 16-bit, 32-bit unsigned integer

Condition codes:
Unaffected

Program exceptions:
None

Description:
The RAISE instruction shall cause an exception with the specified exception code A.

Store Exception Code

ECODE 6A, A

Operand types:
A 8-bit, 16-bit, 32-bit unsigned integer

Condition codes:
Unaffected

Program exceptions:
None

Description:
The ECODE instruction allows an exception handler to obtain the exception code corresponding to the last exception. If the state of the procedure's exception handler is *exception code available*, the exception code shall be stored in the location specified by the operand A. If the procedure's exception handler is in any other state the value stored is undefined. Execution of the ECODE instruction shall place the procedure's exception handler in the *disabled* state.

Set Exception Handler Entry Address

EXCEPT 68, A

Operand types:
A Address Operand

Condition codes:
Unaffected

Program exceptions:
None

Description:
The EXCEPT instruction allows a procedure to redefine the starting address of its exception handler. Execution of the EXCEPT instruction shall place the procedure's exception handler in the *handler defined* state. The handler start address shall be set to the address specified by the operand A.

Exception Return

ERET 6C, A

Operand types:
A 8-bit, 16-bit, 32-bit unsigned integer

Condition codes:
Unaffected

Program exceptions:
Task.Failure

Description:
The ERET instruction returns an exception to the caller of a procedure. The exception code shall be specified by the operand of the instruction. This instruction functions similarly to the RET instruction, except that if there is no caller (BASE = 1) an error condition exists:

BASE = 0 The current procedure context shall be removed from the context stack. The caller's context (which is now at the top of the stack) shall be restored. Bits 16:31 of the PSW shall be restored to the value specified in the caller's context. Bits 0:15 of the PSW shall be unchanged. An exception shall be raised in the restored procedure context with the exception code specified by the ERET instruction.

BASE = 1 The current procedure context shall be removed from the context stack. The supervisor exception handler (section 9.5) shall be invoked with the exception **Task.Failure**. The BASE bit in the supervisor exception handler's context shall be set.

Exception Return and Propagate

ERP 6D, A

Operand types:
A 8-bit, 16-bit, 32-bit unsigned integer

Condition codes:
Unaffected

Program exceptions:
Task.Failure

Description:
The ERP instruction allows the Supervisor Exception Handler to force an exception back to the handler of the procedure in which it occurs. This is done by supressing the examination of the UDLE bit when the exception is first raised. The ERP instruction shall function identically with the ERET instruction except the UDLE bit shall be treated as 0 in the procedure to which the return is made. Should the exception be propagated to the procedure's caller, the UDLE bit of the caller shall function normally.

25. List Instructions

Insert in Doubly Linked List

ILIST 78, Entry, Pred

If all memory accesses can be completed then

(1) (Entry) <- (Pred)
(2) (Entry + 4) <- Pred
(3) ((Pred) + 4) <- Entry
(4) (Pred) <- Entry

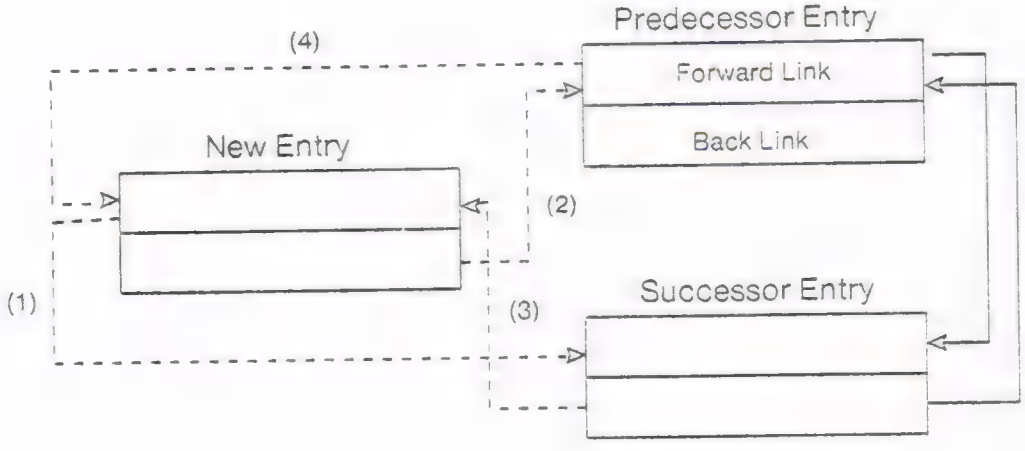
else take memory management fault.

Operand types:
Entry, Pred Address Operands

Condition codes:
Unaffected

Program exceptions:
None

Description:
After checking to ensure that all memory accesses can be completed without a fault, the Entry is placed in the doubly linked list after Pred. A memory management fault will result in retry or leave the operands unaffected, depending on the fault.



Remove from Doubly Linked List

RLIST 79, Entry, Addr

If all memory access can be completed then

(1) ((entry + 4)) <- (Entry)
(2) ((Entry) + 4) <- (Entry + 4)
(3) Addr <- Entry

else take memory management fault.

Operand types:

Entry	Address Operand
Addr	8-bit, 16-bit, 32-bit logical

Condition codes:

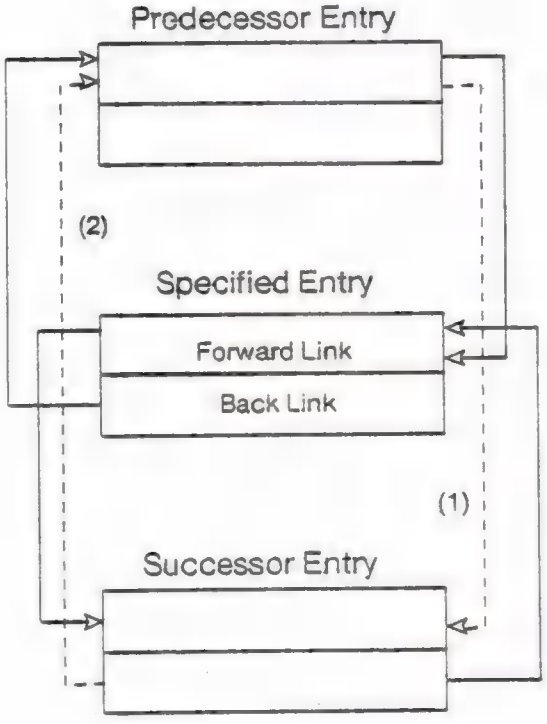
Unaffected

Program exceptions:

None

Description:

After checking to ensure that all memory accesses can be completed without a fault, Entry is removed from a doubly linked list and a pointer to Entry is placed in Addr.



26. String Instructions

Compare Block

CMPBK 95, Cnt, S1, S2

Operand types:

S1, S2	Address Operands
Cnt	8-bit, 16-bit, 32-bit unsigned integer

Condition codes:

Z <- S1 Eql S2
N <- 0
T <- 0
C Unaffected

Program exceptions:

Operand.size

Description:

Strings S1 and S2, each with Cnt items, are compared item by item. If they are identical, the Z bit is set. Otherwise the Z bit is cleared. If size(S1) is different from size(S2) then an **Operand.size** exception shall be raised.

Move Block

MOVBK 94, Cnt, Src, Dest

Operand types:

Src, Dest	Address Operands
Cnt	8-bit, 16-bit, 32-bit unsigned integer

Condition codes:

Unaffected

Program exceptions:

Operand.Size

Description:

Cnt items are copied from (Src) to (Dest). If size(Src) = size(Dest) then overlapping source and destination fields do not affect the results. If size(Src) is different from size(Dest), an **Operand.Size** exception shall be raised. This instruction shall be interruptable.

Move Multiple (Fill)

MOVM 93, Cnt, Src, Dest

Operand types:

Cnt	8-bit, 16-bit, 32-bit unsigned integer
Src	8-bit, 16-bit, 32-bit logical
Dest	Address Operand

Condition codes:

Unaffected

Program exceptions:

None

Description:

Replace Cnt items, beginning at Dest with Src. If size(Src) is larger than size(Dest), Src is truncated and no indication is given (as with other logicals). If size(Src) is smaller than size(Dest), Src is zero extended. This instruction shall be interruptable.

Move Translated

MOVTR 96, Table, Cnt, Src, Dest

Operand types:

Table, Src, Dest	Address Operands
Cnt	8-bit, 16-bit, 32-bit unsigned integer

Condition codes:

Unaffected

Program exceptions:

Operand.size

Description:

Cnt *bytes* are copied from the source string (Src), translated, and placed in the destination string (Dest). Table is the address of a 256 byte vector. Bytes are read from the source string, used to index the table, and the corresponding entry in the table is copied into the destination string. If the source and destination strings overlap, the destination string will contain the correct result. If either size(Src) or size(Dest) is NOT byte, an Operand.size exception shall be raised.

Scan and Break

SCANB 97, Btable, Src, Slen, Dest, Dlen

Operand types:

Btable, Src, Dest — Address Operands
Slen, Dlen — 8-bit, 16-bit, 32-bit unsigned integers

Condition codes:

Z <- Dlen Eql Slen
N <- Dlen Lssu Slen
T Unaffected
C Unaffected

Program exceptions:

Operand.Size

Description:

Bytes are moved from the source string begin at Src, to the destination string beginning at Dest. Btable is the address of a 256-bit vector called the break table. Bytes taken from the source string are used to index the break table. If the bit referenced by the source string byte is reset (0), the source string byte is copied into the destination string and the next byte in the source string is checked. If the bit in the vector is set (1) the byte is NOT copied, the total number of bytes moved is placed in the operand Dlen, and the instruction is terminated.

If Src = Dest, the operation of the instruction is unaffected. If the source and destination strings overlap in other ways, or if the destination string overlaps the break table, the result is undefined.

27. Bit Field Instructions

A bit field is defined by three parameters, the base (Base), the position (Pos) and the size (Size). The base is a byte address, the position is the count in bits from the beginning of the byte referenced by the base to the start of the bit field, and the size is the size of the field. If the specified size is greater than 32 an Illegal.Size exception shall be raised. For references to single bits, the size is not included and is presumed to be one.

Store Bit Field

SBF 7D, Src, Pos, Size, Base

field(Pos,Size,Base) <- Src

Operand types:

Src, Size	8-bit, 16-bit, 32-bit unsigned integer
Pos	8-bit, 16-bit, 32-bit signed integer
Base	Address Operand

Condition codes:

Unaffected

Program exceptions:

None

Description:

The bit field defined by Pos, Size and Base is replaced by the operand Src. If Src is smaller than Size, the high order part of the bit field is filled with zeros.

Load Bit Field (Sign Extended)

LBFS 7E, Pos, Size, Base, R

R <- field(Pos,Size,Base) ; sign extended

Operand types:

Size	8-bit, 16-bit, 32-bit unsigned integer
Pos, R	8-bit, 16-bit, 32-bit signed integer
Base	Address Operand

Condition codes:

Z <- R Eql 0
N <- R Lss 0
T Set if information lost during truncation, otherwise cleared
C Unaffected

Program exceptions:

None

Description:

The contents of the bit field defined by Pos, Size and Base are copied into R. If R is larger than Size, the field is sign extended. If R is smaller than size, the field is truncated from the most significant part and T is set if information is lost.

Load Bit Field (Zero Extended)

LBF 7F, Pos, Size, Base, R

R <- field(Pos,Size,Base) : zero extended

Operand types:

Size	8-bit, 16-bit, 32-bit unsigned integer
Pos	8-bit, 16-bit, 32-bit signed integer
Base	Address Operand
R	8-bit, 16-bit, 32-bit logical

Condition codes:

Z <- R Eql 0
N <- R Lss 0
T Unaffected
C Unaffected

Program exceptions:

None

Description:

The contents of the bit field defined by Pos, Size and Base are copied into R. If R is larger than Size, the field is zero extended. If R is smaller than size, the field is truncated from the most significant part.

Set Bit

SETBIT 74, Pos, Base

N <- Bit(Pos,Base) next
bit(Pos,Base) <- 1

Operand types:

Pos	8-bit, 16-bit, 32-bit signed integer
Base	Address Operand

Condition codes:

Z <- not N
N Set based on original contents of bit
T Unaffected
C Unaffected

Program exceptions:

None

Description:

The bit specified by Pos and Base is interrogated and set to One.

*This operation MUST be performed using a read-modify-write (or interlocked) memory operation on the bit to enable the instruction to be used for synchronization.

Clear Bit

CLRBIT 75, Pos, Base

N <- Bit(Pos,Base) next
Bit(Pos,Base) <- 0

Operand types:

Pos	8-bit, 16-bit, 32-bit signed integer
Base	Address Operand

Condition codes:

Z <- not N
N Set based on original contents of bit
T Unaffected
C Unaffected

Program exceptions:

None

Description:

The bit selected by Pos and Base is interrogated and set to zero.

*This operation MUST be performed using a read-modify-write (or interlocked) memory operation on the bit to enable the Instruction to be used for synchronization.

Invert Bit

INVBIT 76, Pos, Base

N <- Bit(Pos,Base) next
Bit(Pos,Base) <- not Bit(Pos,Base)

Operand types:

Pos	8-bit, 16-bit, 32-bit signed integer
Base	Address Operand

Condition codes:

Z <- not N
N Set based on original contents of bit
T Unaffected
C Unaffected

Program exceptions:

None

Description:

The bit selected by Pos and Base is interrogated and inverted.

*This operation MUST be performed using a read-modify-write (or interlocked) memory operation on the bit to enable the Instruction to be used for synchronization.

Test Bit

TSTBIT 77, Pos, Base

Operand types:

Pos	8-bit, 16-bit, 32-bit signed integer
Base	Address Operand

Condition codes:

N <- Bit(Pos,Base)
Z <- not N
T Unaffected
C Unaffected

Program exceptions:

None

Description:

The bit selected by Pos and Base is interrogated to set the condition codes.

28. Miscellaneous Instructions

No Operation

NOP 6E

Operand types:
None

Condition codes:
Unaffected

Program exceptions:
None

Description:
No operation is performed. The program counter is set to point to the next instruction.

Break

BREAK 6F

Operand types:
None

Condition codes:
Unaffected

Program exceptions:
None

Description:
The supervisor exception handler is invoked with exception code Break.

Load PSW

LPSW 08, A

Operand types:
A 8-bit, 16-bit, 32-bit logical

Condition codes:
Based on the new PSW

Program exceptions:
Privileged Instruction Trap

Description:
The LPSW instruction replaces the caller's PSW contents by the value of its operand. This shall be a privileged instruction. The effect of altering the BASE (PSW<16>), Maxreg (PSW<20:23>), or number of parameters (PSW<24:31>) fields is unpredictable. If the BASE bit of the current PSW = 1 a **Context.Base** exception shall be raised.

Store PSW

SPSW 09, A

Operand types:
A 8-bit, 16-bit, 32-bit logical

Condition codes:
Unaffected

Program exceptions:
Privileged Instruction Trap

Description:
The SPSW instruction shall store the contents of the caller's PSW in the location specified by its operand. If the BASE bit of the current PSW = 1 a **Context.Base** exception shall be raised.

Size

SIZE R <- size(A) 7B, A, R

Operand types:
A, R 8-bit, 16-bit, 32-bit logical

Condition codes:
Z <- R Eql 0
N <- R Lss 0
T Unaffected
C Unaffected

Program exceptions:
None

Description:
The size, in bytes, of the operand A is returned in R.

Set Condition Codes

SETCC CC <- A 7A, A

Operand types:
A 8-bit, 16-bit, 32-bit logical

Condition codes:
The condition codes EAE,C,T,N,Z are set based on the low order bits of A.

Program exceptions:
None

Description:
The condition codes are set based on A.

Window into the Micromachine

WINDOW 0D, Byte of information

Operand types:
None

Condition codes:
Implementation dependent

Program exceptions:
None

Description:
If a console is in place and enabled, the byte of information following the opcode in the instruction stream is passed into the micromachine in an implementation dependent manner. Otherwise the byte of information is skipped and the instruction is treated as a nop.

Replace Entry in Map

REPENT 0A, A, Map, Seg

Operand types:
A, Map, Seg 8-bit, 16-bit, 32-bit logical

Condition codes:
Unaffected

Program exceptions:
Privileged Instruction Trap
Specification.Error

Description:
The replace entry in map instruction provides a guaranteed way of altering an entry in a memory map. Execution of this instruction shall cause the value of the operand A to be stored in the map entry specified by the segment number (Seg) and map pointer number (Map) operands. The supervisor map pointer shall be map number 1; The user map pointer shall be map number 0. If Map specifys a value other than 0 or 1, a Specification.Error exception shall be raised. This instruction shall have the implementation dependent effect of forcing any translation cache to be consistent with this new entry an the current map length stored in memory.

Map Virtual Address

MAP 0B, A, Phy, Seg

Operand types:
A, Phy, Seg 8-bit, 16-bit, 32-bit logical

Condition codes:
Unaffected

Program exceptions:
Privileged Instruction Trap

Description:
The first operand is zero extended to 32-bits and treated as a virtual address. The virtual address is translated using the map specified by bit 0 of the virtual address (after zero extension) and the corresponding physical address is placed in Phy. The associated segment number is placed in Seg.

29. Floating-Point Arithmetic

The floating-point instructions are used to perform calculations on operands with a wide range of magnitude. All floating-point instructions conform to the proposed IEEE standard for binary floating-point arithmetic.[1]

29.1. Floating-point Data Classes. There are five floating-point data classes: normalized numbers, denormalized numbers, normal zeroes, infinities and NaN.

Normalized numbers are any numbers whose exponents are not equal to the formats minimum or the formats maximum.

Denormalized numbers have the format's minimum exponent and a non-zero fraction.

Normal zeroes have a zero exponent and a zero fraction. Both plus and minus zero are allowed.

Infinity has the format's maximum exponent and a zero fraction.

A number with the maximum exponent and a non-zero fraction is a NaN (Not a Number). There are two types of NaN's, trapping and non-trapping. A trapping NaN will create an Illegal.Operation condition when used as a source operand. Non-trapping NaN's will propagate through arithmetic operations without raising exceptions. Trapping NaN's are distinguished by having their most significant fraction bit set. If the number is a NaN and bit 12 in double precision or bit 9 in single precision is set then the number is a trapping NaN.

29.2. Floating-point Formats. Floating-point data shall occupy a fixed-length format which may be either a 32-bit format (single) or a 64-bit format (double). Figure 29-1 shows the number representation for the two formats.

In the single precision format the leftmost bit (bit 0) is the sign bit. Bits 1:8 are occupied by the exponent. The exponent is a biased exponent with a bias of 127. The remaining 23 bits contain the fraction.

In the double precision format the leftmost bit (bit 0) is the sign bit. Bits 1:11 contain the exponent. The exponent is a biased exponent with a bias of 1023. The remaining 52 bits contain the fraction.

[1] J. Coonen et al., "A Proposed Standard for Binary Floating Point Arithmetic", SIGNUM Newsletter, October 1979

	SINGLE	DOUBLE
Fields and widths in bits:		
S = Sign	1	1
E = Exponent	8	11
L = Leading bit (Implicit)	(1)	(1)
F = Fraction	23	52
Total Width	(1)+32	(1)+64
Sign: +/- represented by 0/1 respectively		
Exponent: biased integer		
Max E	255	2047
Min E	0	0
Bias of E	127	1023
Normalized numbers:		
Range of E	(Min E + 1) to (Max E - 1)	
Represented number	$(-1)^S \cdot 2^{E-Bias} \cdot (L.F)$	
Signed zeros:		
E	Min E	Min E
L	(0)	(0)
F	0	0
Reserved Operands:		
Denormalized numbers:		
E	Min E	Min E
L	(0)	(0)
F	nonzero	nonzero
Represented number	$(-1)^S \cdot 2^{E-Bias} \cdot (L.F)$	
Signed ∞'s:		
E	Max E	Max E
L	(0)	(0)
F	0	0
NaN's:		
E	Max E	Max E
L	(0)	(0)
F	nonzero	nonzero

Figure 29-1: Floating Point Number Representation

29.3. Floating-Point Exceptions. There are five maskable floating-point exception conditions. These conditions are **Illegal.Operation**, **Divide.By.Zero**, **Floating.Underflow**, **Floating.Overflow** and **Floating.Inexact**. They are explained further below.

One non-maskable exception condition can occur during execution of the floating-point compare (CMPF) instruction. This exception condition, named **unordered**, shall occur when the operands in the compare cannot be related using conventional relational operators (=,<,>...). An example of this would be a comparison between NaN and a finite number.

Associated with each maskable exception is a "sticky" flag and a mask bit. The sticky flags are bits 27:31 (See Figure 29-2) in the Auxilliary Status Register (ASR). The mask bits are bits 19:23 in the ASR. Bit 27 is the sticky flag and bit 19 is the mask for the Illegal.Operation exception condition. Bits 28 and 20 are the corresponding bits for Divide.By.Zero exception. Bits 29 and 21 are associated with the Floating.Underflow exception. Bits 30 and 22 are associated with the Floating.Overflow exception and bits 31 and 23 are associated with the Floating.Inexact exception.

When an exception condition exists an exception shall be initiated if arithmetic exceptions are

Illegal.Operation (I)
Divide.by.Zero (Q)
Floating.Overflow (O)
Floating.Underflow (U)
Floating.Inexact (P)

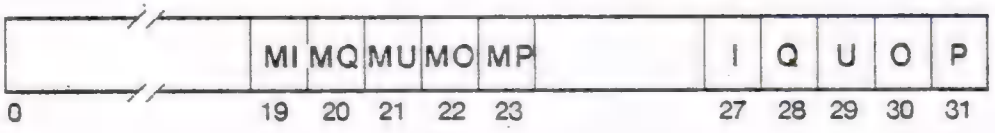


Figure 29-2: Exception Bits in ASR

enabled (EAE is one) and the mask bit for the corresponding exception condition is zero. If EAE is zero then the exception shall be suppressed and Truncate (T) shall be set. If the mask bit is one, the exception shall be suppressed and the corresponding sticky flag shall be set to one. Sticky flags are cleared by moving zeros into the ASR.

29.3.1. Illegal.Operation. The Illegal.Operation condition exists in floating-point arithmetic operations on errors that are not frequent enough or important enough to merit their own fault conditions. An **Illegal.Operation** exception shall be raised if arithmetic exceptions are enabled and the mask bit (bit 19 in the ASR) is zero. If the exception is masked then the sticky flag for this exception (bit 25 in the ASR) shall be set and the result shall be a non-trapping NaN.

There are two cases for Illegal.Operation conditions. The first arises if an operand is illegal for the operation to be performed. The other arises if the result is illegal for the destination.

Some examples of Illegal.Operations are:

$\sqrt{-5}$
$\infty - \infty$
$0 \times \infty$
Storing an unnormalized number

29.3.2. Divide.By.Zero. The Divide.By.Zero condition exists in a division operation when the divisor is normal zero and the dividend is a finite nonzero number. The **Divide.By.Zero** exception shall be raised if arithmetic exceptions are enabled and the mask bit (bit 20 in the ASR) is zero. If the exception is masked then the sticky flag for this condition (bit 26 in the ASR) shall be set and the result shall be infinity with sign according to convention.

29.3.3. Floating.Underflow. A Floating.Underflow condition exists if the exponent of a result, tested before rounding, lies below the exponent range of the destination field. The **Floating.Underflow** exception shall be raised if arithmetic exceptions are enabled and the mask bit (bit 21 in the ASR) is zero. If the exception is masked then the sticky flag for this condition (bit 27 in the ASR) shall be set and the result shall be denormalized by shifting the fraction right while incrementing the exponent until the exponent reaches its minumum allowable value.

29.3.4. Floating.Overflow. If the exponent of a rounded result of an arithmetic operations overflows the range of the destination, then the Floating.Overflow condition exists unless the result is unnormalized in which case an Illegal.Operation condition exists. The Floating.Overflow exception shall be raised if arithmetic exceptions are enabled and the mask bit (bit 22 in the ASR) is zero. If the exception is masked then the sticky flag for this condition (bit 26 in the ASR) shall be set and the result depends on the rounding mode and the sign of the result. The result is set to infinity with the sign of the result if the rounding mode is RN, RZ, RP and the result is positive or RM and the result is negative. Otherwise the result is the largest magnitude normalized number representable in the destination.

29.3.5. Floating.Inexact. If the result cannot be exactly specified in the destination format and no other exceptions are raised a Floating.Inexact condition exists. If arithmetic exceptions are enabled and the mask bit for this exception (bit 23 in the ASR) is zero then the Floating.Inexact exception shall be raised. If the exception is masked then the sticky flag for this condition (bit 29 in the ASR) shall be set and the result shall be the correctly rounded number.

29.4. Rounding. Four rounding modes are supported in the IEEE standard. The four modes are:

RN	-	Round to Nearest
RZ	-	Round toward Zero
RP	-	Round toward + infinity
RM	-	Round toward - infinity

Any instruction returning a result rounds the result according to bits 25 and 26 in the ASR.

Bits 25:26 in ASR	Rounding mode
00	RN
01	RZ
10	RP
11	RM

Preliminary results can be viewed as having been computed to infinite precision. From the preliminary result (R), determine the two numbers (R1 and R2) in the desired format that most closely bracket R.

If R1 = R = R2, there is no rounding error and RN(R) = RZ(R) = RP(R) = RM(R). Otherwise Floating.Inexact is signaled and the value returned is determined from the table below, assuming that R1 < R < R2.

Rounding Mode	Result
RN	the nearer of R1 and R2. If equidistant then the value whose least significant bit is Zero.
RZ	the smaller of R1 and R2 in magnitude.
RP	R2
RM	R1

29.5. Infinity Arithmetic. The IEEE Standard supports two modes of infinity arithmetic. The first mode, affine, specifies $-\infty < +\infty$. In projective mode $-\infty = +\infty$. The mode for infinity arithmetic is specified by bit 18 in the ASR. Zero specifies projective mode; One specifies affine mode. The default mode for infinity arithmetic is projective.

29.6. Floating-Point Instructions. All floating-point instructions follow the same steps during execution. These steps are:

1. Fetch source operands - if either operand is a trapping NaN then raise Illegal.Operation exception.
2. Compute preliminary result.
3. Check Floating.Underflow.
4. Round.
5. Check Floating.Inexact and Floating.Overflow.
6. Return result.

The compare instruction (CMPF) is the only floating point instruction that does not return a result so it only goes through Step 1 above.

Floating-Point Addition

ADDF	R <- A + B	80, A, B, R
	R <- R + A	81, A, R

Operand types:
A, B, R 32-bit, 64-bit floating point

Condition Codes:
Z <- R Eql 0
N <- R Lss 0
T Unaffected
C Unaffected

Program Exceptions:
Illegal.Operation
Floating.Underflow
Floating.Overflow
Floating.Inexact

Description:
The first operand is added to the second operand. In two operand addition the result is placed in the second operand. In three operand addition the result is placed in the third operand. If either operand was normalized the result will be also normalized unless Floating.Underflow occurs.

An Illegal.Operation condition exists whenever either operand is a trapping NaN, when the operation is $(+\infty) + (-\infty)$ or $(-\infty) + (+\infty)$ in Affine mode, or when both operands are infinity in Projective mode. If the mask bit for Illegal.Operation is set (bit 19 in the ASR) then the result is a non-trapping NaN.

A Floating.Underflow condition exists when the intermediate result's exponent is less than the destination format's minumum. If no exception is raised, (ie. the mask bit is one), then the intermediate result is denormalized (fraction shifted right and exponent incremented by one for each bit shift) until the exponent is at the destination format's minumum.

A Floating.Overflow condition exists if the result's exponent value would be greater than or equal to the maximum exponent value of the destination format and the result is not infinity or NaN. If the exception mask bit is one (bit 21 in the ASR) the result depends on the rounding mode and the sign of the result. The result is set to infinity with the sign of the result if the rounding mode is RN, RZ, RP and the result is positive or RM and the result is negative. Otherwise the result is the largest magnitude normalized number representable in the destination.

A Floating.Inexact condition exists if the delivered result is not exactly equal to the intermediate result and no other exceptions have been raised.

Operands may be either single or double floating-point numbers.

Special Cases:

When a result is to be returned and either operand is a NaN then the result will be the same NaN. If both operands are NaNs then the result will be the first operand.

If both operands are zero then the result is +0 in rounding modes RN, RZ, RP, or if both operands are +0. The result is -0 in mode RM or if both operands are -0.

If one operand is infinity then the result is the same infinity.

In Affine mode, $(+\infty) + (+\infty) = (+\infty)$ and $(-\infty) + (-\infty) = (-\infty)$. An Illegal.Operation condition exists if the operation is $(+\infty) + (-\infty)$ or $(-\infty) + (+\infty)$.

In Projective mode if both operands are infinity then an Illegal.Operation condition exists.

Floating-Point Subtraction

SUBF	R <- B - A	82, A, B, R
	R <- R - A	83, A, R

Operand types:

A, B, R 32-bit, 64-bit floating point

Condition Codes:

Z <- R Eql 0
N <- R Lss 0
T Unaffected
C Unaffected

Program Exceptions:

Illegal.Operation
Floating.Underflow
Floating.Overflow
Floating.Inexact

Description:

The first operand is subtracted from the second operand. In the two operand version the result is placed in the second operand. In the three operand version the result is placed in the third operand.

The execution of the subtract operation is identical to that of the add operation except that the sign bit of the first operand is first inverted.

Operands may be either single or double floating-point numbers.

Special Cases:

See ADDF

Floating-Point Multiplication

MULF	R <- B x A	84, A, B, R
	R <- R x A	85, A, R

Operand types:

A, B, R 32-bit, 64-bit floating point

Condition Codes:

Z <- R Eql 0
N <- R Lss 0
T Unaffected
C Unaffected

Program Exceptions:

Illegal.Operation
Floating.Underflow
Floating.Overflow
Floating.Inexact

Description:

The second operand is multiplied by the first operand. In two operand multiplication the result is placed in the second operand. In the three operand operation the result is placed in the third operand.

An Illegal.Operation condition exists whenever either operand is a trapping NaN or when one operand is 0 and the other is infinity. If the mask bit for the exception (bit 19 in the ASR) is set then the result is a non-trapping NaN.

A Floating.Underflow condition exists when the intermediate result's exponent is less than the destination format's minumum. If no exception is raised, (ie. the mask bit is one), then the intermediate result is denormalized (fraction shifted right and exponent incremented by one for each bit shift) until the exponent is at the destination format's minumum.

A Floating.Overflow condition exists if the result's exponent value would be greater than or equal to the maximum exponent value of the destination format and the result is not infinity or NaN. If the exception mask bit is one (bit 21 in the ASR) the result depends on the rounding mode and the sign of the result. The result is set to infinity with the sign of the result if the rounding mode is RN, RZ, RP and the result is positive or RM and the result is negative. Otherwise the result is the largest magnitude normalized number representable in the destination.

A Floating.Inexact condition exists if the delivered result is not exactly equal to the intermediate result and no other exceptions have been raised.

Operands may be either single or double floating-point numbers.

Special Cases:

When a result is to be returned and either operand is a NaN then the result will be the same NaN. If both operands are NaNs then the result will be the first operand.

If one operand is zero and the other is a finite number or if both operands are zero then the result is zero with the proper sign.

If one operand is zero and the other is infinity then an Illegal.Operation condition exists. If no exception is raised the result is a non-trapping NaN.

If either operand is infinity and the other is a finite number or both operands are infinity, then the result is infinity with the sign bit equal to the exclusive OR of the operands' sign bits.

Floating-Point Division

DIVF	R <- B / A	86, A, B, R
	R <- R / A	87, A, R

Operand types:

A, B, R — 32-bit, 64-bit floating point

Condition Codes:

Z <- R Eql 0
N <- R Lss 0
T Unaffected
C Unaffected

Program Exceptions:

Illegal.Operation
Divide.By.Zero
Floating.Underflow
Floating.Overflow
Floating.Inexact

Description:

The second operand (the dividend) is divided by the first operand (the divisor). In two operand division the result is placed in the second operand. In the three operand operation the result is placed in the third operand.

An Illegal.Operation condition exists when either operand is a trapping NaN, when both operands are zero, when both operands are infinity or when the divisor is an unnormalized number. If the mask bit for the exception (bit 19 in the ASR) is set then the result is a non-trapping NaN.

A Divide.By.Zero condition exists if the divisor is zero and the dividend is a non-zero finite number. If no exception is raised (bit 20 in the ASR is one) then the result is infinity with the proper sign.

A Floating.Underflow condition exists when the intermediate result's exponent is less than the destination format's minumum. If no exception is raised, (ie. the mask bit is one), then the intermediate result is denormalized (fraction shifted right and exponent incremented by one for each bit shift) until the exponent is at the destination format's minumum.

A Floating.Overflow condition exists if the result's exponent value would be greater than or equal to the maximum exponent value of the destination format and the result is not infinity or NaN. If the exception mask bit is one (bit 21 in the ASR) the result depends on the rounding mode and the sign of the result. The result is set to infinity with the sign of the result if the rounding mode is RN, RZ, RP and the result is positive or RM and the result is negative. Otherwise the result is the largest magnitude normalized number representable in the destination.

A Floating.Inexact condition exists if the delivered result is not exactly equal to the intermediate result and no other exceptions have been raised.

Operands may be either single or double floating-point numbers.

Special Cases:

When a result is to be returned and either operand is a NaN then the result wil be the same NaN. If both operands are NaNs then the result will be the first operand.

If both operands are zero or both operands are infinity then an Illegal.Operation condition exists and the result is a non-trapping NaN.

If the dividend is zero and the divisor is not then the result is zero with the proper sign.

If the divisor is 0 and the dividend is a non-zero finite number then A Divide.By.Zero condition exists and the result is infinity with the proper sign.

If the dividend is infinity and the divisor is not infinity or NaN then the result is infinity with the proper sign.

Negate Floating

NEGF	R <- (-A)	8A, A, R
	R <- (-R)	8B, R

Operand types:

A, R 32-bit, 64-bit floating point

Condition Codes:

Z <- R Eql 0
N <- R Lss 0
T Unaffected
C Unaffected

Program Exceptions:

Illegal.Operation
Floating.Underflow
Floating.Overflow
Floating.Inexact

Description:

In the two operand version the first operand is placed in the second operand with the sign is negated. In the single operand version the sign bit of the operand is negated.

An Illegal.Operation condition exists whenever the source operand is a trapping NaN. If the mask bit for Illegal.Operation is set (bit 19 in the ASR) then the result is a non-trapping NaN.

A Floating.Underflow condition exists when the number to be put in the second operand has an exponent less than the destination format's minumum. This can only happen if the source operand is a double floating number and the destination is single floating. If no exception is raised, (ie. the mask bit is one), then the intermediate result is denormalized (fraction shifted right and exponent incremented by one for each bit shift) until the exponent is at the destination format's minumum.

An Floating.Overflow condition exists if the result to be put in the destination has an exponent value greater than or equal to the maximum exponent value of the destination format and the result is not infinity or NaN. This can only happen if the source operand is a double floating number and the destination is single floating. If the mask bit is one (bit 21 in the ASR) the result depends on the rounding mode and the sign of the result. The result is set to infinity with the sign of the result if the rounding mode is RN, RZ, RP and the result is positive or RM and the result is negative. Otherwise the result is the largest normalized number representable in the destination with the sign of the intermediate result.

A Floating.Inexact condition exists if the delivered result is not exactly equal to the intermediate result and no other exceptions have been raised. This can only happen if the source operand is a double floating number and the destination is single floating.

Operands may be either single or double floating-point numbers.

Special Cases:

If the first operand is a NaN the sign bit will be inverted.

Convert Integer to Floating

FLOAT R <- A 8C, A, R

Operand types:

A	8-bit, 16-bit, 32-bit signed integer
R	32-bit, 64-bit floating point

Condition Codes:

Z <- R Eql 0
N <- R Lss 0
T Unaffected
C Unaffected

Program Exceptions:

Floating.Inexact

Description:

The source operand is converted from a two's-complement integer format to a floating-point format and placed in the destination operand.

A Floating.Inexact condition exists if the destination does not have exactly the same value as the source. This can only happen when the source is a 32-bit integer and the destination is a 32-bit floating number.

The source operand is a two's-complement integer in byte, 16-bit or 32-bit size. The destination operand must be a 32-bit or 64-bit floating-point format.

Special Cases:

None

Convert Floating to Integer

FIX R <- Integerized(A) 8D, A, R

Operand types:

A	32-bit, 64-bit floating point
R	8-bit, 16-bit, 32-bit signed integer

Condition Codes:

Z <- R Eql 0
N <- R Lss 0
T <- set if the result cannot be correctly represented in R
C Unaffected

Program Exceptions:

Illegal.Operation

Description:

The source operand is converted from a floating-point format to an integer format and placed in the destination operand. The floating-point number is rounded to an integer value according to the rounding mode in effect (bits 25 and 26 in the ASR). The source operand is not affected.

An Illegal.Operation condition exists if the source operand is a NaN or Infinity. The destination operand is left unchanged.

The source operand can be a single or double floating-point number. The destination operand is an 8-bit, 16-bit or 32-bit signed integer.

Special Cases:

If the source operand is a NaN or Infinity an Illegal.Operation condition exists. The destination operand is not affected.

If the result overflows the destination field, then excessive high-order buts are truncated and the T condition code is set.

Move Floating

MOVF B <- A 8E, A, B

Operand types:
A, B 32-bit, 64-bit floating point

Condition Codes:
Z <- B Eql 0
N <- B Lss 0
T Unaffected
C Unaffected

Program Exceptions:
Illegal.Operation
Floating.Underflow
Floating.Overflow
Floating.Inexact

Description:
The first operand is placed in the second operand. The first operand is not affected. This instruction can perform size conversion.

An Illegal.Operation condition exists if the source operand is a trapping NaN. If no exception is raised a non-trapping NaN is placed in the destination operand.

A Floating.Underflow condition can occur if the source operand is double floating and the destination operand is single floating. If no exception is raised the number is denormalized (fraction shifted right and exponent incremented by one for each bit shift) until the exponent is equal to the destination format's minumum.

A Floating.Overflow condition can occur if the source operand is double floating, the destination operand is single floating and the source operand's exponent is greater than or equal to the destination formats maximum and the number is not Infinity or Nan. If no exception is raised the result depends on the rounding mode and the sign of the result. The result is set to infinity with the sign of the result if the rounding mode is RN, RZ, RP and the result is positive or RM and the result is negative. Otherwise the result is the largest magnitude normalized number representable in the destination.

A Floating.Inexact condition exists if the destination is not exactly equal to the source and no other exceptions have been raised.

Both operands are either 32-bit or 64-bit floating-point numbers.

Special Cases:
If the source operand is a trapping NaN and no exception is raised then a non-trapping NaN is placed in the destination operand.

If the operands are of different sizes then size conversion is implied.

Clear Floating

CLRF A <- +0 8F, A

Operand type:
A 32-bit, 64-bit floating point

Condition Codes:
Z <- 1
N <- 0
T Unaffected
C Unaffected

Program Exceptions:
None

Description:
The operand is replaced by positive zero.

The operand can be either a 32-bit or 64-bit floating-point number.

Special Cases:
None

Compare Floating

CMPF 90, A, B

Operand type:

A, B 32-bit, 64-bit floating point

Condition Codes:

T Unaffected
C Unaffected
If A and B are unordered then
    the unordered exception is raised.
Otherwise:
    Z <- A Eql B
    N <- A Lss B

Program Exceptions:

Illegal.Operation
Unordered

Description:

The two operands are compared and the condition codes are set according to the comparison. If the operands cannot be related using conventional relational operators then the unordered exception is raised. Figure 29-3 specifies the compare operation.

X vs Y	− ∞ Affine	Finite	+ ∞ Affine	∞ Projective	NaN
- ∞ Affine	=	<	<	N/A	a
Finite	>	b	<	a	a
+ ∞ Affine	>	>	=	N/A	a
∞ Projective	N/A	a	N/A	=	a
NaN	a	a	a	a	a

a: The unordered exception is taken.

b: The result is based on the result of X - Y. Possible exception conditions are suppressed.

Figure 29-3: The Compare Operation

An Illegal.Operation condition exists if the source operand is a trapping NaN. If the Illegal.Operation exception is not raised then the unnordered exception is raised.

The operands can be either 32-bit or 64-bit floating-point numbers.

Special Cases:

If either operand is a NaN and the Illegal.Operation exception is not raised then the unordered exception shall be raised.

If the mode for inifinity arithemetic is projective and either operand is infinity then the unordered exception will be raised. If both operands are infinty then the comparison is equal.

Floating-Point Square Root

SQRTF R <- $\sqrt{A}$ 89, A, R

Operand types:
A, R 32-bit, 64-bit floating point

Condition Codes:
Z <- R Eql 0
N <- R Lss 0
T Unaffected
C Unaffected

Program Exceptions:
Illegal.Operation
Floating.Inexact

Description:
The square-root of the first operand is placed in the second operand. The first operand is not affected.

An Illegal.Operation condition exists if the source operand is a trapping NaN or if the source operand is negative, if the source operand is denormalized or if the source operand is + infinity and the mode is projective. If no exception is raised the result is a NaN.

A Floating.Underflow condition exists when the intermediate result's exponent is less than the destination format's minumum. If no raised is raised, (ie. the mask bit is one), then the intermediate result is denormalized (fraction shifted right and exponent incremented by one for each bit shift) until the exponent is at the destination format's minumum.

A Floating.Overflow condition exists if the result's exponent value would be greater than or equal to the maximum exponent value of the destination format and the result is not infinity or NaN. If the exception mask bit is one (bit 21 in the ASR) the result depends on the rounding mode and the sign of the result. The result is set to infinity with the sign of the result if the rounding mode is RN, RZ, RP and the result is positive or RM and the result is negative. Otherwise the result is the largest magnitude normalized number representable in the destination.

A Floating.Inexact condition exists if the delivered result is not exactly equal to the intermediate result and no other exceptions have been raised.

Operands may be either single or double floating-point numbers.

Special Cases:
When a result is to be returned and the source operand is a NaN then the result will be the same NaN.

The square-root of $+\infty$ in affine mode is $+\infty$. The square root of ∞ in projective mode creates an Illegal.Operation condition.

Absolute Value Floating

ABSF R <- |A| 91, A, R

Operand types:
A, R 32-bit, 64-bit floating point

Condition Codes:
Z <- B Eql 0
N <- B Lss 0
T Unaffected
C Unaffected

Program Exceptions:
Illegal.Operation
Floating.Underflow
Floating.Overflow
Floating.Inexact

Description:
The first operand is placed in the second operand with the sign bit set ot zero. The first operand is not affected.

An Illegal.Operation condition exists if the source operand is a trapping NaN. If no exception is raised a non-trapping NaN is placed in the destination operand.

A Floating.Underflow condition can occur if the source operand is double floating and the destination operand is single floating. If no exception is raised the number is denormalized (fraction shifted right and exponent incremented by one for each bit shift) until the exponent is equal to the destination format's minumum.

A Floating.Overflow condition can occur if the source operand is double floating and the destination operand is single floating. This condition occurs when the source operand's exponent is greater than or equal to the destination formats maximum and the number is not Infinity or Nan. If no exception is raised, the result depends on the rounding mode and the sign of the result. The result is set to infinity with the sign of the result if the rounding mode is RN, RZ, RP and the result is positive or RM and the result is negative. Otherwise the result is the largest magnitude normalized number representable in the destination.

A Floating.Inexact condition exists if the destination is not exactly equal to the source and no other exceptions have been raised.

Both operands are either 32-bit or 64-bit floating-point numbers.

Special Cases:
If the source operand is a trapping NaN and no exception is raised then a non-trapping NaN, with the sign bit zero, is placed in the destination operand.

If the operands are of different sizes then size conversion is implied.

Remainder Floating

REMF R <- B REM A 88, A, B, R

Operand type:

A, B, R 32-bit, 64-bit floating point

Condition Codes:

Z <- R Eql 0
N <- R Lss 0
T Unaffected
C Unaffected

Program Exceptions:

Illegal.Operation
Floating.Underflow
Floating.Overflow
Floating.Inexact

Description:

The result of the operation B REM A is placed in the destination operand. The remainder is defined regardless of the rounding mode by the following relation when A NEQ 0:

R = B - (A x n) where n is the integer nearest B / A. If
|n - B / A| = ½ then n is even.

An Illegal.Operation condition exists if:

1. both source operands are zero.
2. the dividend is a finite number and the divisor is zero.
3. the dividend is infinity and the divisor is zero.
4. the dividend is infinity and the divisor is a finite number.
5. both source operands are infinity.
6. the divisor is unnormalized.
7. either source operand is a trapping NaN.

If the exception is masked then the result is a non-trapping NaN.

A Floating.Underflow condition exists when the intermediate result's exponent is less than the destination format's minumum. If no exception is raised, (ie. the mask bit is one), then the intermediate result is denormalized (fraction shifted right and exponent incremented by one for each bit shift) until the exponent is at the destination format's minumum.

A Floating.Overflow condition exists if the result's exponent value would be greater than or equal to the maximum exponent value of the destination format and the result is not infinity or NaN.If the exception mask bit is one (bit 21 in the ASR) the result depends on the rounding mode and the sign of the result. The result is set to infinity with the sign of the result if the rounding mode is RN, RZ, RP and the result is positive or RM and the result is negative. Otherwise the result is the largest magnitude normalized number representable in the destination.

A Floating.Inexact condition exists if the delivered result is not exactly equal to the intermediate result and no other exceptions have been raised.

Operands may be either single or double floating-point numbers.

Special Cases:

If the divisor is zero or the dividend is infinity and either operand is a NaN then an Illegal.Operation condition exists.

If the dividend is zero and the divisor is not zero or NaN the result is zero.

If the divisor is infinity and the dividend is a finite number then the result is the dividend.

If either operand is a NaN and no exception is raised the result is the same NaN. If both operands are NaN's then the result is the second operand.

Round to Integer

RNDI R <- A 92, A, R

Operand types:

A, R 32-bit, 64-bit floating point

Condition Codes:

Z <- B Eql 0
N <- B Lss 0
T Unaffected
C Unaffected

Program Exceptions:

Illegal.Operation
Floating.Overflow
Floating.Inexact

Description:

The first operand is rounded to an integer value and then placed in the second operand. The first operand is not affected. If the source's exponents is so large that there are no (zero or non-zero) significant fractions bits, the result will be set to the source.

An Illegal.Operation condition exists if the source operand is a trapping NaN. If no exception is raised a non-trapping NaN is placed in the destination operand.

A Floating.Overflow condition can occur if the source operand is double floating, the destination operand is single floating and the source operand's exponent is greater than or equal to the destination formats maximum and the number is not Infinity or NaN. If no exception is raised the result depends on the rounding mode and the sign of the result. The result is set to infinity with the sign of the result if the rounding mode is RN, RZ, RP and the result is positive or RM and the result is negative. Otherwise the result is the largest magnitude normalized number representable in the destination.

A Floating.Inexact condition exists if the destination is not exactly equal to the source and no other exceptions have been raised.

Both operands are either 32-bit or 64-bit floating-point numbers.

Special Cases:

If the source is zero, infinity, or NaN the destination will be set to the same value.

30. Undefined Opcodes

All opcodes other than those defined in previous sections shall be undefined. Undefined opcodes (OPcode EXceptions or OPEXs) shall cause a vector call using the opcode as an index through the OPEX vector registers. Refer to section 8.5.

I. ISP DESCRIPTION

```
Nebula :=
BEGIN
**Machine.State**

!Processor Status Word

Kernel<> :=       PSW<0>,                 !Kernel/ User mode
Last.mode<> :=    PSW<1>,                 !Previous context (Kernel/Task)
Pri<0:4> :=       PSW<4:8>,               !Processor priority
C<> :=            PSW<9>,                 !Carry condition code
T<> :=            PSW<10>,                !Truncate cc
N<> :=            PSW<11>,                !Negative (less) cc
Z<> :=            PSW<12>,                !Zero cc
Debug<0:1> :=     PSW<13:14>,             !Debugging Control
Privilege<> :=    PSW<15>,                !Privileged if set
Base<> :=         PSW<16>,                !Base of context stack
Superv<> :=       PSW<17>,                !Supervisor/Task mode
UDLE<> :=         PSW<18>,                !Up/Down level exception
EAE<> :=          PSW<19>,                !Exception on Arithmetic Error
MaxReg<0:3> :=    PSW<20:23>,             !Number of registers in current context
MaxPar<0:7> :=    PSW<24:31>,             !Number of parameters in current context

PC<0:31>,                                 !Program Counter

!Machine State Registers

Ctxp[0:1]<0:31> :=      MSR[0:1]<0:31>,         !Context Pointers
 Kctxp<0:31> :=         Ctxp[0]<0:31>,          !Kernel
 Tctxp<0:31> :=         Ctxp[1]<0:31>,          !Task
Soft.int.req<0:31> :=   MSR[2]<0:31>,           !Software Interrupt Request
PSW<0:31> :=            MSR[3]<0:31>,           !Processor Status
Vreg[0:3]<0:31> :=      MSR[4:7]<0:31>,         !SVC and OPEX vector registers
ASR<0:31> :=            MSR[8]<0:31>,           !Auxillary Status Register
  IC<> :=               ASR<18>,                !Infinity Control
  MI<> :=               ASR<19>,                !Mask for Invalid Operand
  MQ<> :=               ASR<20>,                !Mask for Division by Zero
  MO<> :=               ASR<21>,                !Mask for Overflow
  MU<> :=               ASR<22>,                !Mask for Underflow
  MP<> :=               ASR<23>,                !Mask for Inexact
  RC<0:1> :=            ASR<25:26>,             !Rounding Control
  I<> :=                ASR<27>,                !Floating Pt. Invalid operand
  Q<> :=                ASR<28>,                !Floating Pt. Division by Zero
  O<> :=                ASR<29>,                !Floating Pt. Overflow
  U<> :=                ASR<30>,                !Floating Pt. Underflow
  P<> :=                ASR<31>,                !Floating Pt. Inexact
MMreg[0:1]<0:31> :=     MSR[9:10]<0:31>,        !Memory management registers
Timctl<0:31> :=         MSR[11]<0:31>,          !Timer Control Reg
Inttim[0:3]<0:31> :=    MSR[12:15]<0:31>,       !Interval timers
TOD<0:31> :=            MSR[16]<0:31>,          !Time of Day
```

```
!
!Exception codes
!

Macro Spec.error :=       |1|,
Macro Ill.Mode :=         |2|,
Macro Ill.Param :=        |3|,
Macro Ill.Reg :=          |4|,
Macro Ill.Write :=        |5|,
Macro Ill.Size  :=        |6|,
Macro Ill.Addr  :=        |7|,
Macro Bad.displacement:=|8|,
Macro Context.Alignment := |9|,
Macro Context.Base :=     |10|,

!Arithmetic Group
Macro Ill.Divisor :=      |16|,
Macro Truncate :=         |17|,
Macro Range.error :=      |18|,
Macro Ill.Operation :=    |19|,
Macro Div.by.Zero :=      |20|,
Macro Overflow :=         |21|,
Macro Underflow :=        |22|,
Macro Inexact :=          |23|,
Macro Unordered :=        |24|,


Macro Task.Failure :=     |32|,
Macro Break :=            |33|,
Macro Inst.Trace :=       |34|,
Macro Proc.Trace :=       |35|,

!
!Vectors
!

!Physical
Macro    SI.vec :=        |"4|,    !Software interrupt vector
Macro    PI.vec :=        |"8|,    !Parameterized Interrupts
Macro    MM.vec :=        |"C|,    !Memory Management Errors
Macro    ME.vec :=        |"10|,   !Memory system errors
                          !Hard error "14, Soft error "10
Macro    PF.vec :=        |"18|,   !Power failure
Macro    PR.vec :=        |"1C|,   !Power restore
Macro    Kernel.Save :=   |"20|,   !Pointer to kernel save area
Macro    Exception.vec:=  |"24|,   !Supervisor exception handler
Macro    Priv.error :=    |"28|,   !Privileged instruction in user mode
!Macro Timer0.Vec :=      |"30|,   !Timer 0 Vector
!Macro Timer1.Vec :=      |"34|,   !Timer 1 Vector
!Macro Timer2.Vec :=      |"38|,   !Timer 2 Vector
!Macro Timer3.Vec :=      |"3C|,   !Timer 3 Vector
!?!The following macros define the values of operand type and
```

```
!?!size used in the ISP implementation.

!?!Operand Sizes (op.addr<Op.Size>)
!Note that these are visible as the size fields of operands.
Macro   Dwrd := |'00|,                  !8 bytes
Macro   Byte := |'01|,                  !1 byte
Macro   Hwrd := |'10|,                  !2 bytes
Macro   Word := |'11|,                  !4 bytes

!?!Operand Types (op.addr<Op.Type>)
Macro   Constant := |'00|,              !?!Descriptor IS the operand
Macro   Context := |'01|,               !?!Register or in context stack
Macro   Literal := |'10|,               !?!Literal in code stream
Macro   Memory  := |'11|,               !?!In Memory

Macro   single := |0|,                  !single size
Macro   double := |1|,                  !Double size

!Size converts an operand size into the equivalent number of bytes
Size(ops<1:0>)<4:0> :=
        BEGIN
        DECODE ops =>
                BEGIN
                Dwrd := size = 8,
                Byte := size = 1,
                Hwrd := size = 2,
                Word := size = 4
                END
        END,

!Sign extend a value VAL of size OPS to 64 bits
sxt(val<63:0>,ops<1:0>)<63:0> :=
        BEGIN
        DECODE ops =>
                BEGIN
                Dwrd := sxt = val,
                Byte := sxt <= val<7:0>,
                Hwrd := sxt <= val<15:0>,
                Word := sxt <= val<31:0>
                END
        END,

Macro   Op.type := |35:34|,
Macro   Op.size := |33:32|,
**Memory.Access** {US}

Macro MaxMem := |4095|,                 ! Model Dependent

!Main Memory

Mb[0:MaxMem]<0:7>,
Mw[0:MaxMem]<0:31>{increment: 4} :=      Mb[0:MaxMem]<0:7>,
```

```
Md[0:MaxMem]<0:63>{increment: 8} :=      Mb[0:MaxMem]<0:7>,

!I/O Space

!MBio["FFF00000:"FFFFFFFF]<0:7>,          !Full I/O space is 20 bits physical

MACRO IO.Min := |"FFFFF000|,     !Small amount defined for ISP's purpose

!I/O space may be accessed on ALIGNED ADDRESSES as byte, half, word, double

MBio[IO.Min:"FFFFFFFF]<0:7>,
MHio[IO.Min:"FFFFFFFF]<0:15>{increment: 2} := MBio[IO.Min:"FFFFFFFF]<0:7>,
MWio[IO.Min:"FFFFFFFF]<0:31>{increment:4}:= MBio[IO.Min:"FFFFFFFF]<0:7>,
MDio[IO.Min:"FFFFFFFF]<0:63>{increment:8}:= MBio[IO.Min:"FFFFFFFF]<0:7>,

MSR[0:16]<0:31> :=       MBio["FFFFF800:"FFFFF843]<0:7>,
!Memory read routine
read(va<31:0>,ops<1:0>,a<2:0>)<63:0> :=
    BEGIN
    DECODE vp(va,a)<0:11> EQL "FFF =>
        BEGIN
        !Memory
        DECODE ops =>
            BEGIN
        Dwrd := read = Mb[vp]@ Mb[vp(va+1,a)]@ Mb[vp(va+2,a)]@ Mb[vp(va+3,a)]@
                Mb[vp(va+4,a)]@ Mb[vp(va+5,a)]@ Mb[vp(va+6,a)]@ Mb[vp(va+7,a)],
        Byte := read = Mb[vp],
        Hwrd := read = Mb[vp] @ Mb[vp(va+1,a)],
        Word := read = Mb[vp] @ Mb[vp(va+1,a)] @ Mb[vp(va+2,a)] @ Mb[vp(va+3,a)]
            END,
        !I/O Space
        DECODE ops =>
            BEGIN
        Dwrd := read = MDio[vp],
        Byte := read = MBio[vp],
        Hwrd := read = MHio[vp],
        Word := read = MWio[vp]
            END
        END
    END,

!Memory write routine
write(va<31:0>,ops<1:0>,a<2:0>)<63:0> :=
    BEGIN
    DECODE vp(va,a)<0:11> EQL "FFF =>
        BEGIN
        !Memory
        DECODE ops =>
            BEGIN
        Dwrd := Mb[vp]@ Mb[vp(va+1,a)]@ Mb[vp(va+2,a)]@ Mb[vp(va+3,a)]@
                Mb[vp(va+4,a)]@Mb[vp(va+5,a)]@Mb[vp(va+6,a)]@Mb[vp(va+7,a)]=write,
        Byte := Mb[vp] = write,
```

```
        Hwrd := Mb[vp] @ Mb[vp(va+1,a)] = write,
        Word := Mb[vp]@ Mb[vp(va+1,a)]@ Mb[vp(va+2,a)]@ Mb[vp(va+3,a)] = write
            END,
        !I/O Space
        DECODE ops =>
            BEGIN
        Dwrd := MDio[vp] = write,
        Byte := MBio[vp] = write,
        Hwrd := MHio[vp] = write,
        Word := MWio[vp] = write
            END
        END
   END,
**Address.Translation**{us}

Mptr[0:1]<0:31> :=      MMreg[0:1]<0:31>,       !Map address registers

!Fields in Mptr registers
MACRO Map.addr :=       |1:28|,         !address of map (bits 29:31 are 0)
!Bit 29 is reserved
MACRO Map.reloc :=      |30|,           !set if relocation enabled
MACRO Map.prot :=       |31|,           !set if protection enabled

M.ent<0:63>,    !Map entry temporary
!Map entry fields
M.priv<> :=             M.ent<0>,       !Privilege
M.bound<0:27> :=        M.ent<1:28>,    !Virtual Address Bound
M.prot<0:2> :=          M.ent<29:31>,   !Protection Key
M.reloc<0:28> :=        M.ent<32:60>,   !Relocation Amount

Macro M.Maxp := |4|,    !Implementation dependent seg. number size
Seg.Max<0:M.Maxp>,      !Maximum Seg number temporary
M.addr<0:31>,           !Map address temporary


!Memory Access Codes
MACRO M.c :=    |0|,    !Context Access
MACRO M.i :=    |1|,    !Instruction fetch
MACRO M.r :=    |2|,    !Memory read
MACRO M.w :=    |3|,    !Memory read/write
MACRO M.n :=    |4|,    !No Access

!Fault error codes
MACRO Inv.sup :=        |1|,    !Invalid access to supervisor space
MACRO Inv.Seg :=        |2|,    !No Seg containing this address
MACRO Inv.accs :=       |3|,    !Access type violation
MACRO Priv.violation := |4|,

!
!Virtual to physical Address translation
!
vp(va<0:31>,mode<0:2>)<0:31> :=
```

```
        BEGIN
        vp = va NEXT
        IF va<0> AND NOT Superv=> M.fault(Inv.Sup) NEXT !User access to superv

        IF Mptr[Va<0>]<30:31> => Seg.number(va) NEXT

        IF Mptr[Va<0>]<Map.Prot> =>     !Protection Enabled
            BEGIN
                IF M.Ent<M.priv> AND NOT Privilege=> M.fault(Priv.Violation)NEXT
                DECODE mode =>
                    BEGIN
                M.c:=   IF M.prot NEQ '011 => M.fault(Inv.accs),
                M.i:=   IF M.prot<0:1> NEQ '10 => M.fault(Inv.accs),
                M.r:=   IF M.prot<1:2> NEQ '01 AND
                                M.prot NEQ '010 => M.fault(Inv.accs),
                M.w:=   IF M.prot NEQ '010 => M.fault(Inv.accs),
                M.n:=   No.Op()
                    END
            END NEXT

        DECODE Mptr[Va<0>]<Map.Reloc> =>
            BEGIN
                vp<0:4> <={tc} va<0> AND va<5:11> EQL "7F, !Fix for I/O Space
                vp = va<5:31> + M.reloc@'000            !Relocation Enabled
            END
        END,

Seg.number(va<0:31>)<0:M.Maxp> :=
        BEGIN
        Seg.number = 0;
        M.addr = Mptr[va<0>]<Map.addr>@'000 NEXT
        Seg.max = Mw[M.addr-4] NEXT
        REPEAT BEGIN
            M.ent = Md[M.addr] NEXT     !Get map entry
            IF va<1:29> LEQ M.bound => LEAVE Seg.number NEXT
            Seg.number = Seg.number + 1;
            M.addr = M.addr + 8 NEXT
            IF Seg.number GTR Seg.max => M.fault(Inv.Seg)
            END
        END,
M.fault(fcode<0:15>)<0:15> :=
        BEGIN
        M.fault = fcode; MMf = 1 NEXT   !Set fault
        RESTART Run                     !And abort
        END,
**Context.Stack.Access**  {US}

! Most of this section is implementation dependent

!The context stack contains the context of the currently running process.
!The top of the stack contains the current process registers.
!This stack is restricted to word (4 byte) boundaries.
```

```
!?!In most implementations the top of stack will be cached in some way.
!?!The ISP reflects one very simple mechanism.


!Context stack read

Read.Ctx(disp<31:0>)<31:0> :=
        BEGIN
        DECODE disp =>
                BEGIN
                0 :=    Read.Ctx = PC,          ! PC is cached in the ISP
                Otherwise :=    Read.Ctx = Mw[vp(Ctxp[Kernel] + disp@'00, M.c)]
                END
        END,

! Context Stack Write

Write.Ctx(Disp<31:0>)<31:0> :=
        BEGIN
        Mw[vp(Ctxp[Kernel] + Disp@'00, M.c)] = Write.Ctx
        END,

Reload.Ctx<31:0> :=
        BEGIN
        PC = Mw[vp(Ctxp[kernel], M.c)] NEXT
        Reload.ctx = Mw[vp(Ctxp[Kernel]-4, M.c)]
        END,
**Call.mechanism**{us}

!
!Call.switch implements the procedure call mechanism with parameter passing
!

P.Var<> :=              Call.Switch<15>,
P.Exception<> :=        Call.Switch<14>,
MaxReg.New<3:0> :=      Call.Switch<11:8>,
MaxPar.New<7:0> :=      Call.Switch<7:0>,
pcount<31:0>,                           !Parameter counter temp
Ctx.size<31:0>;                         !New context size temp

Call.switch(addr<31:0>)<15:0> :=
        BEGIN
        call.switch = read(addr,Hwrd,M.i) NEXT  !get procedure descriptor
        addr = addr + 2;

        !Determine number of parameters

        IF P.Var =>                     ! variable number of parameters?
                BEGIN
                MaxPar.New = get.log() NEXT
                IF get.log GTR 255 => Exception(Ill.Param)
                END NEXT
```

```
        !Determine size of new context

        Ctx.size = Maxpar.new + Maxreg.New + 3 NEXT
        write.ctx("FFFFFFFF) = PSW NEXT          ! Save current PSW

        !Set exception handler for this context

        DECODE P.Exception =>
                BEGIN
                0 :=    write.ctx = 0,
                1 :=    BEGIN
                        write.ctx = read(addr,Hwrd,M.i) + addr NEXT
                        addr = addr + 2
                        END
                END NEXT
        write.Ctx("FFFFFFFE) NEXT                ! Save Exception Handler

        !Evaluate the Parameters

        pcount = 0 NEXT
        loop := REPEAT
                BEGIN
                IF pcount EQL MaxPar.New => LEAVE loop NEXT
                IF op.addr()<op.type> EQL Context =>     !Ref to prior register
                        op.addr<31:0> = op.addr<31:0> + Ctx.size NEXT
                Write.ctx(pcount-Maxpar.new-2)=op.addr<35:31>@op.addr<26:0> NEXT
                pcount = pcount + 1
                END NEXT

        !complete the control transfer

        write.ctx(0) = PC NEXT           ! Save current PC
        !check existance to -Ctx.size -1 (new PSW storage)
        Ctxp[Kernel] = Ctxp[Kernel] - Ctx.size@'00 NEXT        ! Point to new cont
        IF MaxReg GTR 0 AND MaxReg.New GTR 0 =>          ! Copy "SP"
                write.ctx(1) = read.ctx(Ctx.size+1) NEXT         !if both exist
        PC = addr;                       ! New PC
        PSW<18:31> = Call.Switch;        ! New PSW
        Base = 0; Superv = Superv AND addr<31>
        END,
!
!Build.stack initiates a call stack on the current stack
!

!descriptor is same format as for Call.stack
B.exception<> :=        Build.stack<14>,
B.maxreg<3:0> :=        Build.stack<11:8>,

Build.stack(inc<31:0>,addr<31:0>,np<7:0>)<15:0> :=
        BEGIN
        Build.stack = read(addr,Hwrd,M.i) NEXT
```

```
        addr = addr + 2;
        Build.stack<7:0> = np;  !Number of parameters
        Ctxp[Kernel] = Ctxp[Kernel] - (B.maxreg + np + 3 - inc)@'00 NEXT

        !install exception handler

        DECODE B.exception =>
            BEGIN
                write.ctx = 0,  !none
                BEGIN
                    write.ctx = read(addr,hwrd,M.i) + addr NEXT
                    addr = addr + 2
                END
            END NEXT
        Write.ctx(B.maxreg+np+1) NEXT

        !Set psw

        PSW<18:31> = Build.stack; Superv = addr<31>;Debug = 0;
        PC = addr
        END,

!
!Call.restore removes a context frame from the context stack
!

Call.restore :=
        BEGIN
        Ctxp[Kernel] = Ctxp[Kernel] + (Maxreg + MaxPar + 3)@'00 NEXT
        DECODE Base =>
            BEGIN
                psw<13:31> = Reload.ctx(),
                BEGIN
                    Kernel = last.mode NEXT
                    psw = Reload.ctx()
                END
            END
        END,

!
!Pop.stack removes an entire execution stack from the context stack
!
Pop.stack :=
        BEGIN
        REPEAT BEGIN
            Ctxp[Kernel] = Ctxp[Kernel] + (Maxreg+MaxPar+3)@'00 NEXT
            IF Base => LEAVE Pop.stack NEXT
            PSW = Read.ctx("FFFFFFFF)
            END
        END,
**Operand.Descriptors** {US}
```

```
!Operand descriptor routine
!Reads an operand specifier from instruction stream and generates
!A descriptor specifying its type, size, and address

op.addr<35:0> :=
BEGIN
**Local.Declarations**

op.spec<7:0>,                           !operand specifier

Parameter(Num<7:0>)<35:0> :=
        BEGIN
        IF Num GTR{us} MaxPar => Exception(Ill.param) NEXT
        DECODE Num NEQ 0 =>
                BEGIN
                0\F :=  parameter = MaxPar,
                1\T :=  BEGIN
                        Read.Ctx(Num + Maxreg) NEXT
                        Parameter = Read.Ctx<31:27>@"0@Read.Ctx<26:0>
                        END
                END
        END,

op.decode<7:0> :=
        BEGIN
        op.spec = read(pc,byte,M.i) NEXT
        pc = pc + 1 NEXT

        DECODE op.spec =>
                BEGIN
                                        !Short Literal Mode
        '000????? :=    op.addr = op.spec,      !Constant

        '0010???? :=    BEGIN           !Register Mode
                        IF op.spec<3:0> EQL 0 OR op.spec<3:0> GTR Maxreg =>
                                        Exception(Ill.Reg) NEXT
                        op.addr = Context@Word@"0000000@op.spec<3:0>
                        END,

                                        !Short Parameter Mode
        '00110??? :=    op.addr = parameter(op.spec<2:0>),

        '001110?? :=    BEGIN           !Recursive Modes (parameter, indexed)
                        IF op.decode NEQ 0 => Exception(Ill.Mode) NEXT
                        op.decode = op.spec     !Save for recursive evaluation
                        END,

        '001111?? :=    BEGIN           !Literal Mode
                        op.addr = Literal@op.spec<1:0>@pc NEXT
                        pc = pc + size(op.spec<1:0>)
                        END,
```

```
        '??000000 :=    BEGIN           !Absolute Address
                        op.addr= Memory@op.spec<7:6>@read(pc,Word,M.i)<31:0>NEXT
                        pc = pc + 4
                        END,

        '??00???? :=    BEGIN           !Register Indirect
                        IF op.spec<3:0> GTR Maxreg => Exception(Ill.Reg) NEXT
                        op.addr = Memory@op.spec<7:6>@Read.Ctx(op.spec<3:0>)
                        END,

        OTHERWISE :=    BEGIN           !Indexed Memory Modes
                        IF op.spec<3:0> GTR Maxreg => Exception(Ill.Reg) NEXT
                        read(pc,op.spec<5:4>,M.i) NEXT  !Index
                        sxt(read,op.spec<5:4>) NEXT     !SignExtend
                        op.addr<35:32> = Memory@op.spec<7:6>;
                        op.addr<31:0> = sxt + Read.ctx(op.spec<3:0>) NEXT
                        pc = pc + size(op.spec<5:4>)
                        END
                END
        END,

MAIN entry :=
        BEGIN
        op.decode = 0 NEXT
        IF op.decode() EQL 0 => LEAVE op.addr NEXT      !Non-recursive modes
        DECODE op.decode() =>
                BEGIN
        '00111000 :=    BEGIN   !Parameter
                        IF fetch(op.addr,0)<63:8> => Exception(Ill.param) NEXT
                        op.addr = parameter(fetch)
                        END,

        '00111001 :=    BEGIN   !Unscaled Index
                        fetch(op.addr,1) NEXT   !Index
                        op.decode() NEXT        !Base
                        op.addr<31:0> = op.addr + fetch
                        END,

        '00111010 :=    BEGIN   !Scaled Index Single Length
                        fetch(op.addr,1) NEXT   !Index
                        op.decode() NEXT        !Base
                        op.addr<31:0>=op.addr + fetch*size(op.addr<Op.Size>)
                        END,

        '00111011 :=    BEGIN   !Scaled Index Double Length
                        fetch(op.addr,1) NEXT   !Index
                        op.decode() NEXT        !Base
                        op.addr<31:0>=op.addr + fetch*size(op.addr<Op.Size>)*2
                        END
                END
```

```
        END
END,

**Access.by.Descriptor**

!fetch operand
!op.adr - descriptor of operand
!s - set for sign extend

fetch(op.adr<35:0>,s<>)<63:0> :=
        BEGIN
        DECODE op.adr<Op.type> =>
                BEGIN
                Constant:= Fetch = op.adr<31:0>,
                Context := Fetch = Read.Ctx(op.adr),
                Literal := Fetch = read(op.adr,op.adr<Op.size>,M.i),
                Memory := Fetch = read(op.adr,op.adr<Op.size>,M.r)
                END NEXT
        IF s => fetch = sxt(fetch,op.adr<op.size>)
        END,

Store(op.adr<35:0>)<63:0> :=
        BEGIN
        DECODE op.adr<op.type> =>
                BEGIN
                Constant := Exception(Ill.write),
                Context := Write.Ctx(op.adr) = store,
                Literal := Exception(Ill.write),
                Memory  := Write(op.adr,op.adr<op.size>,M.w) = store
                END
        END,
**Operand.Access**

Get.int(d<>)<63:0> :=
        BEGIN
        op.addr() NEXT
        IF d AND op.addr<op.type> EQL Memory =>
                op.addr<op.size> = op.addr<op.size> + 1 NEXT    !double size
        get.int = fetch(op.addr,1)
        END,

Put.int(d<>,repl<>)<63:0> :=
        BEGIN
        IF NOT repl =>
                BEGIN
                op.addr() NEXT
                IF d AND op.addr<op.type> EQL Memory =>
                        op.addr<op.size> = op.addr<op.size> + 1
                END NEXT
        Store(op.addr) = put.int;
        t = (put.int NEQ sxt(put.int,op.addr<op.size>)) NEXT
        n = sxt<63>;
```

```
        z = sxt EQL 0
        END,

Get.log<31:0> :=
        BEGIN
        op.addr() NEXT
        get.log = fetch(op.addr,0)
        END,

Put.log(repl<>)<31:0> :=
        BEGIN
        IF NOT repl => op.addr() NEXT
        store(op.addr) = put.log;
        n = sxt(put.log,op.addr<op.size>)<31> NEXT
        z = sxt EQL 0
        END,

Get.float<79:0> :=
         BEGIN
         op.addr() NEXT
         IF op.addr<35:32> EQL literal @ byte =>
                BEGIN
                get.float = special.case() NEXT
                LEAVE get.float
                END NEXT
        IF op.addr<op.type> EQL Memory =>
                op.addr<op.size> = op.addr<op.size> + 1 NEXT
        Get.float = unpack(fetch(op.addr,0),op.addr<op.size>)
        END,

special.case<79:0> :=
        Begin
        !Decode fetch(Op.Addr,0) =>
        !       Begin
        !       special.case = ?,
        !       special.case = ?
                !.....
        !       End
        no.op()
        End,

Put.float(repl<>)<79:0> :=
        BEGIN
        IF NOT repl =>
                BEGIN
                op.addr() NEXT
                IF op.addr<op.type> EQL Memory =>
                                op.addr<op.size> = op.addr<op.size> + 1
                END NEXT
        store(op.addr) = pack(put.float,op.addr<op.size>)
        END,
```

```
Get.field(pos<31:0>,size<31:0>)<31:0> :=
        BEGIN{us}
        IF size GTR 32 => exception(ill.size) NEXT
        IF size EQL 0 => (get.field = 0; LEAVE get.field) NEXT
        op.addr<31:0> = op.addr()<31:0> +{tc} pos<31:3> NEXT    !Byte Address
        Memory.Chk() NEXT
        pos = op.addr<1:0>@pos<2:0> NEXT        !Position from word boundary
        size = size + pos -1 NEXT                !End bit position
        read = MASK.LEFT(read(op.addr AND "FFFFFFFA,Word+size<5>,M.r), pos) NEXT
        get.field = read SR0 (31-size<4:0>)
        END,

Put.field(pos<31:0>, size<31:0>, repl<>)<31:0> :=
        BEGIN{us}
        IF size GTR 32 => exception(ill.size) NEXT
        IF size EQL 0 => LEAVE put.field NEXT
        put.field = MASK.LEFT(put.field, 32-size);
        IF NOT repl =>
            op.addr<31:0> = op.addr()<31:0> +{tc} pos<31:3> NEXT
        Memory.Chk() NEXT
        pos = op.addr<1:0>@pos<2:0> + size -1 NEXT     !position of end bit
        op.addr<1:0> = 0 NEXT
        read = read(op.addr, Word+pos<5>, M.r) SRR (31-pos) NEXT
        read = MASK.RIGHT(read, size) OR put.field NEXT
        write(op.addr, Word+pos<5>, M.w) = read SLR (31-pos)
        END,
** Instruction.Interpretation ** {US}

ir<7:0>,                                !?!instruction register
replace<>       := ir<0>,               ! Result in last operand fetched
long.branch<>   := ir<0>,               ! 16 bit branch displacement
pc.back<31:0>,                          ! initial PC for fault recovery

Start :=
        BEGIN
        Soft.Int.Req = ASR = Exception = 0;
        !Ctxp = Vreg = MMreg = undefined<31:0>
        Power.up() NEXT
        Run()
        END,

Run :=  BEGIN
        Int.Service() NEXT

        pc.back = pc;
        ir = read(pc,byte,M.i) NEXT
        pc = pc + 1 NEXT
        IEX() NEXT
        RESTART run
        END,

        REQUIRE.ISP |IEX.ISP|,
```

```
!
!Exception handler
!
Exception(ecode<29:0>)<> :=
BEGIN
    DECODE UDLE AND NOT Exception =>
        BEGIN
        0:=    BEGIN    !Upward to calling routines
                PC = read.ctx(maxreg+MaxPar+1) NEXT     !Get specified handler
                IF PC NEQ 0 =>          !If handler exists
                    BEGIN
                        Write.ctx(maxreg+MaxPar+1) = ecode;      !save code
                        LEAVE Iex
                    END NEXT

                IF NOT Base =>          !No handler but caller exists
                    BEGIN
                        Call.restore();
                        RESTART Exception       !Try the caller
                    END NEXT

                !No handler and bottom of stack
                Sup.eh(maxreg+MaxPar+3,Task.Failure) NEXT
                Base = 1        !This is all that's left
            END,
        1:=    Sup.eh(0,ecode)  !Down to the supervisor
        END NEXT
    LEAVE Iex
END,

Sup.eh(soff<0:31>,ecode<29:0>) :=       !Entry to supervisor exception handler
        BEGIN
        Write.ctx("FFFFFFFF) = PSW NEXT
        Write.ctx(0) = PC NEXT
        Superv = 1 NEXT
        Build.Stack(soff,Mw[Exception.vec]<0:29>@'00,3) NEXT
        Privilege = Mw[exception.vec]<31>;
        Write.ctx(Maxreg+1) = ecode NEXT
        Write.ctx(Maxreg+2) = Memory@Byte@pc.back<0>@pc.back<5:31> NEXT
        Write.ctx(Maxreg+3) = Context@(Maxreg+maxpar+3)<29:0>
        END,

OPEX :=         !Unimplemented opcode handler
BEGIN
    vector.call(0,ir)
END,

Vector.call(b<>,index<15:0>)<31:0> :=
BEGIN
    DECODE index LSS Vreg[b@'0]<0:15> =>
        BEGIN
        0\F :=  index = index - Vreg[b@'0]<0:15>,
```

```
        1\T :=  index = 0
        END NEXT
    IF index GTR Vreg[b@'0]<16:31> => index = 0;
    superv = privilege = 1 NEXT !Full privilege for vector access
    Vector.call = Mw[vp( Vreg[b@'1] + index@'00, M.r)] NEXT
    Call.switch(vector.call<31:2>@'00) NEXT
    Privilege = Vector.call<0>; Debug = 0
END,

Fp.exception(ecode<4:0>) :=
    BEGIN
    DECODE ASR<ecode> =>
        BEGIN
        0  :=
            DECODE EAE =>
                BEGIN
                t = 1,
                exception(ecode)
                END,
        1  := ASR<ecode +{US} 8> = 1
        END
    END,

**Interrupt.Service**{us}

!?!A device may request an interrupt by storing its vector location
!?!in Ext.vec and setting the appropriate bit of Ext.int.vec. Note
!?!that this IMPLEMENTATION is for the convenience of the ISP and should
!?!not be taken literally
Ext.int.req<0:31>,              !?!External interrupt request
Ext.vec[0:31]<0:31>,            !?!External interrupt vector

!?!The following are set by the memory system when errors occur.
!?!A soft error will set these only if enabled in the ASR
!?!control register is set
MER<0:31>,                      !?!Address of failed memory location
HME<>,                          !?!Set if hard memory error

MMf<> :=        Int.service<0>, !?!Memory Management Fault
Mem.err<> :=    Int.service<1>, !?!Memory system hard or soft error
Pwr.fail<> :=   Int.service<2>, !?!Power failure

Rp.tmp<0:5>,    !Temp for priority

Int.Service<0:2> :=
        BEGIN
        !Internal interrupts
        IF Int.Service =>
            DECODE first.one(Int.Service) =>
                BEGIN
                0:= BEGIN       !Memory management fault
```

```
                    MMf = 0; trap(MM.vec,4) NEXT
                    !Fill in parameters
                    Write.ctx(Maxreg+1) = Memory@Byte@vp<31>@vp<26:0> NEXT
                    Write.ctx(Maxreg+2) = Memory@Byte@PC.back<31>@
                                                    pc.back<26:0> NEXT
                    Write.ctx(Maxreg+3) = seg.number NEXT   !Segment number
                    Write.ctx(Maxreg+4) = M.fault           !Fault code
                END,
            1:= BEGIN       !Memory error
                    Mem.err = 0; trap(ME.vec+HME@'00,1) NEXT
                    Write.ctx(Maxreg+1) = MER<27:0>
                END,
            2:= BEGIN       !Power fail
                    pwr.fail = 0;
                    !?!Implementation shall flush all caches at this point
                    Mer = Mw[Kernel.save] NEXT
                    Mw[Mer] = Kctxp<0:30>@Kernel;  !Save kernel context
                    Mw[Mer+4] = Mptr[1];    !and supervisor map pointer
                    Trap(PF.vec,0) NEXT
                    Pri = "1F       !Priority to maximum
                END
            END NEXT

        !External Interrupts

        Rp.tmp = last.one(mask.left(Ext.int.req OR Soft.int.req,Pri+1)) NEXT
        IF Rp.tmp EQL 32 => LEAVE Int.service NEXT
        Rp.tmp = 31 - Rp.tmp NEXT   !Convert to request priority
        DECODE last.one(Ext.int.req) LEQ last.one(Soft.int.req) =>
            BEGIN
            0\F :=  BEGIN   !Software interrupt
                    Soft.Int.req<Rp.tmp> = 0;
                    trap(SI.vec,1) NEXT
                    Write.ctx(Maxreg+1) = Rp.tmp
                    END,
            1\T :=  BEGIN   !External Interrupt
                    Ext.int.req<Rp.tmp> = 0;
                    DECODE Mw[Ext.vec[Rp.tmp]]<31> =>
                        BEGIN
                        0:= trap(Ext.vec[Rp.tmp],0),  !Just vector to it
                        1:= BEGIN   !Parameterized Handler
                                trap(PI.vec,0) NEXT
                                IF Maxreg=> write.ctx(1)=Mw[Ext.vec[Rp.tmp]]
                            END
                        END
                    END
            END NEXT
        Pri = Rp.tmp        !Raise priority
    END,

trap(vec<0:31>,nparms<0:7>) :=
    BEGIN
```

```
        Write.ctx(0) = PC NEXT
        Write.ctx("FFFFFFFF) = PSW NEXT
        last.mode = Kernel NEXT
        Kernel = 1 NEXT
        build.stack(0,Mw[vec],nparms) NEXT
        Base = 1
        END,

Power.up :=
        BEGIN
        Mer = Mw[Kernel.save] NEXT
        Kctxp = Mw[Mer] AND "FFFFFFFC; Kernel = Mw[Mer]<31>;
        Mptr[1] = Mw[Mer+4];
        Trap(Mw[PR.vec],0) NEXT
        Pri = "1F
        END,
** Instructions ** {TC}
        tmp<31:0>,                          ! global single precision temporaries
        tmp1<31:0>,
        tmp2<31:0>,
        tmp.d<63:0>,                        ! global double precision temporary
        tmp.t<>,                            ! temporary for truncate info

Privilege.chk :=
        BEGIN
        IF NOT Privilege =>
            BEGIN
                write.ctx("FFFFFFFF) = PSW NEXT
                write.ctx(0) = pc.back NEXT
                superv = 1 NEXT
                build.stack(0,Mw[priv.error],1) NEXT
                write.ctx(maxreg+1) = Memory@Byte@pc.back<31>@pc.back<26:0>
            END
        END,

Memory.Chk :=
        BEGIN
        IF op.addr<Op.Type> NEQ Memory => Exception(Ill.addr)
        END,
! integer add

ADD.ex:=
        BEGIN
        tmp = get.int(single) NEXT              !get first operand
        c@put.int(single,replace) = get.int(single)+tmp
        END,

! integer subtract

SUB.ex:=
        BEGIN
        tmp = get.int(single) NEXT  !get minuend
```

```
        c@put.int(single,replace) = get.int(single) + NOT tmp +{US} 1
        END,

! integer multiply single precision

MUL.ex:=
        BEGIN
        tmp = get.int(single) NEXT              ! get first operand
        put.int(single,replace) = get.int(single)*tmp
        END,

! integer divide single precision without remainder

DIV.ex:=
        BEGIN
        tmp = get.int(single) NEXT              ! get first operand
        IF tmp EQL 0 => Exception(Ill.Divisor) NEXT
        put.int(single,replace) = get.int(single)/tmp
        END,

! integer negate

NEG.ex:=
        BEGIN
        c@put.int(single,replace) = NOT get.int(single) +{US} 1
        END,

! logical NOT

NOT.ex:=
        BEGIN
        put.log(replace) = NOT get.log()
        END,

! integer remainder B over A

REM.ex:=
        BEGIN
        tmp1    = get.int(single) NEXT
        IF tmp1 EQL 0 => Exception(Ill.Divisor) NEXT
        put.int(single,0) = get.int(single) MOD tmp1
        END,

! integer modulus

MOD.ex:=
        BEGIN
        tmp1    = get.int(single) NEXT
        IF tmp1 EQL 0 => Exception(Ill.Divisor) NEXT
        put.int = get.int(single) MOD tmp1 NEXT
        IF get.int<31> XOR tmp1<31> => put.int = put.int + tmp1 NEXT
        put.int(single,0)
```

```
        END,

! integer multiply double precision

EMUL.ex:=                                   ! extended integer multiply
        BEGIN
        tmp = get.int(single) NEXT
        put.int(double,0) = get.int(single)*tmp
        END,

! integer divide double precision with remainder

EDIV.ex:=                           ! extended integer divide (with remainder)
        BEGIN
        tmp = get.int(single) NEXT                  ! get divisor
        IF tmp EQL 0 => Exception(Ill.Divisor) NEXT
        tmp.d = get.int(double) NEXT
        put.int(single,0) = tmp.d MOD tmp NEXT  ! compute rem(B/A)
        tmp.t = t NEXT
        put.int(double,0) = tmp.d/tmp NEXT              ! compute B/A
        t = t OR tmp.t OR ( tmp.d<63> AND tmp<31> AND put.int<63>)
        END,

! integer increment and decrement by fixed constants

!               R = R + nnn
!               R = R - nnn

INC.ex :=       (put.int(single,1) = get.int(single) + 1),      ! increment by 1
INC2.ex :=      (put.int(single,1) = get.int(single) + 2),      ! increment by 2
INC4.ex :=      (put.int(single,1) = get.int(single) + 4),      ! increment by 4
INC8.ex :=      (put.int(single,1) = get.int(single) + 8),

DEC.ex :=       (put.int(single,1) = get.int(single) + "FFFFFFFF),

! integer add single precision with carry in

ADDC.ex:=                                   ! R = B + A + carry
        BEGIN
        tmp = get.int(single) NEXT
        c@put.int(single,0) = get.int(single) + tmp + {US}c
        END,

! integer subtract single precision with carry in

SUBC.ex:=                                   ! R = B + (NOT A) + carry
        BEGIN
        tmp = get.int(single) NEXT      ! get  A
        c@put.int(single,0) = get.int(single) + (NOT tmp) + {us}c
        END,

! sign extended move
```

```
MOV.ex:= (put.int(single,0) = get.int(single)),

! integer compare A with B

CMP.ex:=                                    ! integer compare A with B
        BEGIN
        tmp1    = get.int(single) NEXT
        tmp2    = get.int(single) NEXT
        z = tmp1 EQL tmp2;
        n = tmp1 LSS tmp2;
        t = 0
        END,

! integer compare within bounds

CMPWB.ex:=                          ! compare integer tmp with bounds A and B
        BEGIN
        tmp = get.int(single) NEXT                  ! get variable
        tmp1    = get.int(single) NEXT              ! get 1st bound
        tmp2    = get.int(single) NEXT              ! get 2nd bound
        z = tmp GEQ tmp1  AND tmp LEQ tmp2;
        n = tmp LSS tmp1;
        t = 0
        END,

! range check
RANGE.ex :=
        BEGIN
        tmp = get.int(single) NEXT                  ! get variable
        tmp1    = get.int(single) NEXT              ! get 1st bound
        tmp2    = get.int(single) NEXT              ! get 2nd bound
        IF tmp LSS tmp1 OR tmp GTR tmp2 =>
            exception(Range.error)
        END,

! integer compare A with ZERO

TEST.ex:=                                   ! compare A with 0,
        BEGIN
        get.int(single) NEXT
        z = get.int EQL 0;
        n = get.int LSS 0;
        t = 0
        END,

ABS.ex:=                    !Absolute value
        BEGIN
        IF get.int(single)<0> => get.int = NOT get.int + 1 NEXT
        put.int(single,0) = get.int
        END,
```

```
EQL.ex:=
        BEGIN
        put.log(0) <= z
        END,

NEQ.ex:=
        BEGIN
        put.log(0) <= NOT z
        END,

LSS.ex:=
        BEGIN
        put.log(0) <= n
        END,

GTR.ex:=
        BEGIN
        put.log(0) <= NOT (n OR z)
        END,

LEQ.ex:=
        BEGIN
        put.log(0) <= n OR z
        END,

GEQ.ex:=
        BEGIN
        put.log(0) <= NOT n
        END,


! arithmetic shift left and right single precision

ASH.ex:=                                ! R= B shifted |A| bit positions
                                        ! R = R shifted |A| bit positions
                                        ! if A >= 0:  LEFT shift with zero fill
                                        ! if A < 0:  RIGHT shift with sign fill
                                        ! t = 1 if sign changes during shift
        BEGIN
        tmp1 = get.int(single) NEXT             ! get shift specifier
        tmp2 = get.int(single) NEXT             ! get source & sign extend
        DECODE tmp1<31:5> =>
                BEGIN
                "0000000 :=     BEGIN
                                put.int(single,0) <= tmp2 SL0 tmp1 NEXT
                                t = t OR tmp2 NEQ (put.int SRD tmp1)
                                END,
                "7FFFFFF :=     BEGIN
                                put.int(single,0) = tmp2 SRD (NOT tmp1 + 1)
                                END,
                OTHERWISE := DECODE tmp1<31> =>
                                BEGIN
```

```
                                0:=     BEGIN
                                        put.int(single,0) = 0;
                                        t = tmp2 NEQ 0
                                        END,
                                1:=     BEGIN
                                        put.int(single,0) <= tmp2<31>;
                                        t = 0
                                        END
                                END
                        END
                END,

! logical AND single precision

AND.ex:=
        BEGIN
        tmp = get.log() NEXT
        put.log(replace) = get.log() AND tmp
        END,

! logical OR single precision

OR.ex:=
        BEGIN
        tmp = get.log() NEXT
        put.log(replace) = get.log() OR tmp
        END,

! logical XOR single precision

XOR.ex:=
        BEGIN
        tmp = get.log() NEXT
        put.log(0) = get.log() XOR tmp
        END,

! rotate

ROT.ex :=
        BEGIN
        tmp1 = get.int(single) NEXT
        tmp2 = get.int(single) NEXT
        DECODE tmp1<31> =>
            BEGIN
            DECODE op.addr<op.size> =>
                BEGIN
                Dwrd := put.log(0) = UNDEFINED(),
                Byte := put.log(0) = tmp2<7:0> SLR tmp1<4:0>,
                Hwrd := put.log(0) = tmp2<15:0> SLR tmp1<4:0>,
                Word := put.log(0) = tmp2<31:0> SLR tmp1<4:0>
                END,
            DECODE op.addr<op.size> =>
```

```
                BEGIN
                Dwrd := put.log(0) = UNDEFINED(),
                Byte := put.log(0) = tmp2<7:0> SRR (-tmp1)<4:0>,
                Hwrd := put.log(0) = tmp2<15:0> SRR (-tmp1)<4:0>,
                Word := put.log(0) = tmp2<31:0> SRR (-tmp1)<4:0>
                END
            END
        END,

! Logical Shift

LSH.ex :=
        BEGIN
        tmp1 = get.int(single) NEXT
        tmp2 = get.log() NEXT
        DECODE tmp1<31:5> =>
                BEGIN
                "0000000 :=     put.log(0) = tmp2 SL0 tmp1,
                "7FFFFFF :=     put.log(0) = tmp2 SR0 (NOT tmp1 + 1),
                OTHERWISE :=    put.log(0) = 0
                END
        END,

! logical move

MOVL.ex:= (put.log(0) = get.log()),

EXCH.ex :=
        BEGIN
        tmp = get.log() NEXT                !get A
        tmp.d = op.addr NEXT                !save location of A
        put.log(1) = get.log() NEXT         !get B and insure writable
        store(tmp.d) = get.log NEXT         !write B into A
        put.log(1) = tmp                    !Write A into B
        END,

MOVA.ex:=
        BEGIN
        op.addr() NEXT
        Memory.Chk() NEXT
        put.log(0) = op.addr
        END,

MOVBK.ex:=      !Move Block
        BEGIN
        tmp = get.int(single) NEXT
        op.addr() NEXT
        Memory.Chk() NEXT
        tmp.d = op.addr<31:0> NEXT
        op.addr() NEXT
        Memory.Chk() NEXT
        DECODE tmp.d<31:0> GTR{US} op.addr<31:0> =>
```

```
            BEGIN
        0 :=    BEGIN           ! start from the end
                tmp.d<31:0> = tmp.d<31:0> +{US} tmp;
                op.addr<31:0> = op.addr<31:0> +{US} tmp NEXT
                Repeat  BEGIN
                    IF tmp EQL{US} 0 => LEAVE movbk.ex NEXT
                    store(op.addr) = read(tmp.d<31:0>,op.addr<op.size>,M.r) NEXT
                    tmp = tmp -{US} 1;
                    tmp.d = tmp.d -{US} 1;
                    op.addr<31:0> = op.addr<31:0> -{US} 1
                    END !movbkb.loop
                END,    !decode case 0
        1 :=    Repeat  BEGIN   ! normal direction
                    IF tmp EQL{US} 0 => LEAVE movbk.ex NEXT
                    store(op.addr) = read(tmp.d<31:0>,op.addr<op.size>,M.r) NEXT
                    tmp = tmp -{US} 1;
                    tmp.d = tmp.d +{US} 1;
                    op.addr<31:0> = op.addr<31:0> +{US} 1
                    END,
            END !decode
        END,

MOVM.ex:=
        BEGIN
        tmp = get.int(single) NEXT
        get.log() NEXT
        op.addr() NEXT
        Memory.Chk() NEXT
        Repeat  BEGIN
                IF tmp EQL 0 => LEAVE movm.ex NEXT
                store(op.addr) = get.log NEXT
                tmp = tmp -{US} 1;
                op.addr<31:0> = op.addr<31:0> +{US} 1
                END
        END,


! logical compare zero extended

CMPU.ex:=
        BEGIN
        tmp1 = get.log() NEXT
        tmp2 = get.log() NEXT
        z = tmp1 EQL tmp2;                          ! A=B
        n = tmp1 LSS{US} tmp2;              ! A<B
        t = 0
        END,

! clear operand and condition codes

CLR.ex:=
        BEGIN
```

```
        put.log(0) = 0                  ! clear operand
        END,

JUMP.ex:=
        BEGIN
        op.addr() NEXT
        Memory.Chk() NEXT
        PC = op.addr
        END,

!=============================================================================
!  Floating Point Instructions

REQUIRE.ISP |FLOAT.ISP|,        !Floating arithmetic operators

          ! floating add

addf.ex :=
    BEGIN
        ftmp = get.float() NEXT
        fact = get.float() NEXT
        float.add() NEXT
        put.float(replace) = ftmp
    END,

          ! floating subtract

subf.ex :=
    BEGIN
        fact = get.float() NEXT
        ftmp = get.float() NEXT
        fact<s> = NOT fact<s> NEXT
        float.add() NEXT
        put.float(replace) = ftmp
    END,

!       Floating Multiply

MULF.ex  :=
  BEGIN
  ftmp = get.float() NEXT
  fact = get.float() NEXT
  float.mult() NEXT
  put.float(replace) = ftmp
  END,

!       Floating Divide

DIVF.ex  :=
  BEGIN
  fact = get.float() NEXT
  ftmp = get.float() NEXT
```

```
  float.div() NEXT
  put.float(replace) = ftmp
  END,

!       Negate Floating

NEGF.ex  :=
  BEGIN
  get.float() NEXT
  put.float(replace) = NOT get.float<70> @ get.float<69:0>
  END,

!       Convert integer to floating

FLOAT.ex  :=
  BEGIN
  tmp1 = get.int(0) NEXT
  put.float(0) = int2float(tmp1)
  END,

!       Convert floating to integer

FIX.ex  :=
  BEGIN
  fact = get.float() NEXT
  DECODE (fact<e> EQL "7FF) AND (fact<f> NEQ 0) =>
    BEGIN
    0  :=
      BEGIN
      t = 0 NEXT
      float2int(fact) NEXT
      DECODE t  =>
        BEGIN
        0  := put.int(single,0) = float2int,
        1  :=
          BEGIN
          put.int(single,0) = float2int NEXT
          t = 1
          END
        END
      END,
    1  :=
      BEGIN
      put.int(single,1) = get.int(single) NEXT
      fp.exception(Ill.Operation)
      END
    END
  END,

!       Move floating

MOVF.ex  :=
```

```
  BEGIN
  put.float(0) = get.float()
  END,

!       Clear floating

CLRF.ex  :=
  BEGIN
  put.float(0) = 0
  END,

!       Compare floating

CMPF.ex  :=
  BEGIN
  ftmp = get.float() NEXT
  fact = get.float() NEXT
  float.cmp()
  END,

!       Square Root floating

SQRTF.ex  :=
  BEGIN
  fact = get.float() NEXT
  put.float(0) = fp.sqrt(fact)
  END,

!       Absolute value floating

ABSF.ex  :=
  BEGIN
  put.float(0) = 0 @ get.float()<69:0>
  END,

!       Round floating to integer value

RNDI.ex  :=
  BEGIN
  fact = get.float() NEXT
  put.float(0) = rnd2int(fact)
  END,

!       Floating Remainder

REMF.ex  :=
  BEGIN
  fact = get.float() NEXT
  ftmp = get.float() NEXT
  fp.rem() NEXT
  put.float(0) = ftmp
  END,
```

```
!=============================================================================

branch(condition<>)<1:0> :=                         !common branch routine
BEGIN
branch = long.branch +{US} 1 NEXT       !Size of displacement
DECODE condition =>
        BEGIN
        pc = pc + size(branch),         !NO branch
        pc = pc + sxt(read(pc,branch,M.i),branch)
        END
END,

!Unconditional Branch
BR.ex := (branch(1)),

!Branch on equal
BEQL.ex := (branch(z)),

!Branch not equal
BNEQ.ex := (branch(NOT z)),

!Branch less or equal
BLEQ.ex := (branch( z OR n)),

!Branch on less
BLSS.ex := (branch(n)),

!Branch greater or equal
BGEQ.ex := (branch(NOT n)),

!Branch greater than
BGTR.ex := (branch(NOT (n or z))),

!Branch if carry set
BCS.ex := (branch(c)),

!Branch if carry clear
BCC.ex := (branch(NOT c)),

!Branch if truncate set
BTS.ex := (branch(t)),

!Branch if truncate clear
BTC.ex := (branch(NOT t)),

CASE.ex:=
        BEGIN
        tmp = get.int(single) NEXT
        tmp = tmp - get.int(single) NEXT
        DECODE tmp LSS{US} get.int(single) =>
```

```
                BEGIN
                pc = pc + (get.int * 2),        !Sel exceeds Num - 1
                pc = pc + sxt(read(pc + (tmp * 2),Hwrd,M.i),Hwrd)
                END
        END,

LOOP.ex:=
        BEGIN
        tmp = get.int(single) NEXT      !get increment
        put.int(single,1) = get.int(single) + tmp NEXT  !add to counter
        get.int(single) NEXT    !get limit
        long.branch = 1 NEXT
        branch(((tmp GEQ 0) AND (put.int LEQ get.int)) OR
                ((tmp LSS 0) AND (put.int GEQ get.int)))
        END,

ENTLP.ex:=
        BEGIN
        tmp = get.int(single) NEXT      !get initial counter
        pc = pc + sxt(read(pc,Hwrd,M.i),Hwrd) NEXT  !get disp to loop control
        DECODE read(pc,byte,M.i) =>
                BEGIN
                "27:=   BEGIN                                   ! LOOP
                        pc = pc + 1 NEXT
                        get.int(single) NEXT    !get increment
                        store(op.addr()) = tmp NEXT     !load counter
                        tmp = get.int NEXT      !save increment
                        get.int(single) NEXT    !get limit
                        long.branch = 1 NEXT
                        branch(((tmp GEQ 0) AND (store LEQ get.int)) OR
                                ((tmp LSS 0) AND (store GEQ get.int)))
                        END,
                "2D:=   BEGIN                                   ! IBLEQ
                        pc = pc + 1 NEXT
                        store(op.addr()) = tmp NEXT     !load counter
                        get.int(single) NEXT            !get limit
                        long.branch = 1 NEXT
                        branch(store LEQ get.int)
                        END,
                "29:=   BEGIN                                   ! IBLSS
                        pc = pc + 1 NEXT
                        store(op.addr()) = tmp NEXT     !load counter
                        get.int(single) NEXT            !get limit
                        long.branch = 1 NEXT
                        branch(store LSS get.int)
                        END,
                "2B:=   BEGIN                                   ! DBGEQ
                        pc = pc + 1 NEXT
                        store(op.addr()) = tmp NEXT     !load counter
                        get.int(single) NEXT            !get limit
                        long.branch = 1 NEXT
                        branch(store GEQ get.int)
```

```
                END,
        "2F:=   BEGIN                                   ! DBGTR
                pc = pc + 1 NEXT
                store(op.addr()) = tmp NEXT             !load counter
                get.int(single) NEXT                    !get limit
                long.branch = 1 NEXT
                branch(store GTR get.int)
                END,
        OTHERWISE:= (exception(Bad.Displacement))
        END !of DECODE
    END,

IBLEQ.ex:=
    BEGIN
    store(op.addr) = get.int(single) + 1 NEXT           !increment counter
    get.int(single) NEXT                !get limit
    long.branch = 1 NEXT
    branch(store LEQ get.int)
    END,

IBLSS.ex:=
    BEGIN
    store(op.addr) = get.int(single) + 1 NEXT           !increment counter
    get.int(single) NEXT                !get limit
    long.branch = 1 NEXT
    branch(store LSS get.int)
    END,

DBGEQ.ex:=
    BEGIN
    store(op.addr) = get.int(single) - 1 NEXT           !increment counter
    get.int(single) NEXT                !get limit
    long.branch = 1 NEXT
    branch(store GEQ get.int)
    END,

DBGTR.ex:=
    BEGIN
    store(op.addr) = get.int(single) - 1 NEXT           !increment counter
    get.int(single) NEXT                !get limit
    long.branch = 1 NEXT
    branch(store GTR get.int)
    END,

CALL.ex :=              !Procedure call
    BEGIN
    op.addr() NEXT
    op.addr<30:31> = 0;
    Memory.Chk() NEXT
    call.switch(op.addr) NEXT
    IF ir<0> => Privilege = 0
    END,
```

```
SVC.ex :=
        BEGIN
        get.log() NEXT
        vector.call(1,get.log)
        END,

JSR.ex:=
        BEGIN
        op.addr() NEXT
        Memory.Chk() NEXT
        Write.Ctx(1) = Read.Ctx(1) - 4 NEXT      !SP <- SP-4
        write(Write.Ctx,Word,M.w) = pc NEXT
        pc = op.addr<31:0>
        END,

RSR.ex:=
        BEGIN
        pc = read(Read.Ctx(1),Word,M.r) NEXT
        Write.Ctx(1) = Read.Ctx + 4
        END,

RET.ex :=
        BEGIN
        call.restore()
        END,

ERET.ex :=
        BEGIN
        UDLE = 0; Write.ctx(maxreg+MaxPar+1) = 0 NEXT
        get.log() NEXT
        Exception(get.log)
        END,

ERP.ex :=
        BEGIN
        DECODE Base =>
            BEGIN
                BEGIN   !There is a caller
                    Get.log() NEXT
                    Call.restore() NEXT
                    Exception(get.log) = 1      !force to user handler
                END,
                exception(Context.base) !no caller
            END
        END,

RAISE.ex:=
        BEGIN
        exception(get.log())
        END,
```

```
ECODE.ex :=
        BEGIN
        Put.log(0) = Read.ctx(maxreg+MaxPar+1) NEXT
        Write.ctx(maxreg+MaxPar+1) = 0
        END,

EXCEPT.ex :=
        BEGIN
        op.addr() NEXT
        Memory.Chk() NEXT
        Write.ctx(maxreg+MaxPar+1) = op.addr
        END,

LPSW.ex :=
        BEGIN
        Privilege.Chk() Next
        IF Base => exception(Context.Base) NEXT
        Write.ctx(Maxreg+Maxpar+3) = get.log()
        END,

SPSW.ex :=
        BEGIN
        Privilege.Chk() Next
        IF Base => exception(Context.Base) NEXT
        put.log(0) = Read.ctx(Maxreg+Maxpar+3)
        END,

BREAK.ex :=
        BEGIN
        Sup.eh(0,Break)
        END,
NOP.ex:=
        BEGIN
        NO.OP()
        END,

LTASK.ex :=
        BEGIN
        Privilege.Chk() Next
        Get.int(double) NEXT
        IF get.int<33:32> => exception(Context.Alignment);
        IF get.int<1:0> EQL '10 => exception(spec.error) NEXT
        Tctxp = get.int<63:32>;
        Mptr[0] = get.int<31:0>
        END,

STASK.ex :=
        BEGIN
        Privilege.Chk() Next
        put.int(double,0) = Tctxp@Mptr[0]
        !Flush task context to memory
        END,
```

```
TSTART.ex :=
        BEGIN
        Privilege.Chk() Next
        Write.ctx("FFFFFFFF) = PSW NEXT
        Write.ctx(0) = PC NEXT
        IF get.log()<31:1> NEQ 0 => exception(spec.error) NEXT
        IF ir<0> => pop.stack() NEXT
        Kernel = get.log<0> NEXT
        PC = read.ctx(0) NEXT
        PSW = read.ctx("FFFFFFFF)
        END,

TRAISE.ex :=
        BEGIN
        get.log() NEXT
        Tstart.ex() NEXT
        PC.back = pc NEXT
        Exception(get.log)
        END,

TINIT.ex :=
        BEGIN
        Privilege.Chk() Next
        Write.ctx("FFFFFFFF) = PSW NEXT
        Write.ctx(0) = PC NEXT
        PC = get.log() NEXT
        get.log() NEXT  !get process half of psw
        IF ir<0> => pop.stack() NEXT
        PSW<0:15> = get.log;
        Kernel = NOT Kernel NEXT
        Build.stack(0,PC,0) NEXT
        Base = 1
        END,

SBF.ex:=
        BEGIN
        tmp = get.log() NEXT    !get source
        tmp1 = get.int(single) NEXT     !get position
        put.field(tmp1,get.log(),0) = tmp
        END,

LBFS.ex:=               !Load Bit Field (Sign extended)
        BEGIN{US}
        tmp = get.int(single) NEXT      !get position
        get.field(tmp,get.log()) NEXT
        put.int(single,0) = ((get.field SL0 (32- get.log)) SRD (32- get.log))
        END,

LBF.ex:=                !Load Bit Field
        BEGIN
        tmp = get.int(single) NEXT      !get position
```

```
        put.log(0) = get.field(tmp,get.log())
        END,

SETBIT.ex:=
        BEGIN
        !Operation is interlocked (read-modify-write)
        n = get.field(get.int(single),1) NEXT
        put.field(get.int,1,0) = 1 NEXT
        z = NOT n;
        t = 0
        END,

CLRBIT.ex:=
        BEGIN
        !Operation is interlocked (read-modify-write)
        n = get.field(get.int(single),1) NEXT
        put.field(get.int,1,0) = 0 NEXT
        z = NOT n;
        t = 0
        END,

INVBIT.ex:=
        BEGIN
        !Operation is interlocked (read-modify-write)
        n = get.field(get.int(single),1) NEXT
        put.field(get.int,1,0) = z = NOT n;
        t = 0
        END,

TSTBIT.ex:=             !Test Bit
        BEGIN
        n = get.field(get.int(single),1)<0> NEXT
        z = NOT n;
        t = 0
        END,


PUSH.ex:=               !push onto sp stack
        BEGIN
        IF maxreg EQL 0 => exception(Ill.reg) NEXT
        write.ctx(1) = read.ctx(1) - 4 NEXT
        write(write.ctx,word,M.w) = get.log()
        END,

POP.ex:=                !pop from sp stack
        BEGIN
        IF maxreg EQL 0 => exception(Ill.reg) NEXT
        put.log(0) = read(read.ctx(1),word,M.r) NEXT
        write.ctx(1) = read.ctx + 4
        END,

MTS.ex:=                !Move To Stack
```

```
        BEGIN
        tmp1 = get.log() NEXT
        tmp = op.addr<op.size> NEXT
        store = get.log() - size(tmp) NEXT      !compute new S but don't store
        write(store,tmp,M.w) = tmp1 NEXT
        store(op.addr)                  !store S now
        END,

MFS.ex:=                !Move From Stack
        BEGIN
        tmp = get.log() NEXT
        tmp.d = op.addr NEXT    !save addr of S
        tmp1 = op.addr()<op.size> NEXT  !get size A
        put.log(1) = read(tmp,tmp1,M.r) NEXT    !store A
        store(tmp.d) = tmp + size(tmp1)         !store new S
        END,

ILIST.ex:=              !Insert in doubly linked LIST
        BEGIN
        tmp = op.addr() NEXT    !get entry address (E)
        Memory.Chk() NEXT
        tmp1 = op.addr() NEXT   !get address of entry to insert after (P)
        Memory.Chk() NEXT
        !get address of successor of P (S) and check for write rights
        tmp2 = read(tmp1,word,M.w) NEXT
        read(tmp,word,M.w) NEXT         !check write rights
        read(tmp+4,word,M.w) NEXT       !check write rights
        read(tmp2+4,word,M.w) NEXT      !check write rights
        !IF tmp<1:0> OR tmp1<1:0> OR tmp2<1:0> => exception(Ill.Operand) NEXT
        write(tmp,word,M.w) = tmp2 NEXT !E(fwd) <= S
        write(tmp+4,word,M.w) = tmp1 NEXT       !E(back) <= P
        write(tmp2+4,word,M.w) = tmp NEXT       !S(back) <= E
        write(tmp1,word,M.w) = tmp      !P(fwd) <= E
        END,

RLIST.ex:=              !Remove from doubly linked LIST
        BEGIN
        tmp = op.addr() NEXT    !get entry address E
        Memory.Chk() NEXT
        tmp1 = read(tmp,word,M.r) NEXT  !get address of successor (S)
        tmp2 = read(tmp+4,word,M.r) NEXT        !get address of pred (P)
        read(tmp2,word,M.w) NEXT        !check access rights
        read(tmp1+4,word,M.w) NEXT      !check access rights
        !IF tmp<1:0> OR tmp1<1:0> OR tmp2<1:0> => exception(Ill.Operand) NEXT
        write(tmp2,word,M.w) = tmp1 NEXT        !P(fwd) <= S
        write(tmp1+4,word,M.w) = tmp2 NEXT      !S(back) <= P
        store(op.addr()) = tmp
        END,

MULFIX.ex:=             !Multiply fixed point
        BEGIN{TC}
        tmp = 0 NEXT
```

```
        tmp1 = get.int(single) NEXT
        IF tmp1<31> => (tmp<0> = 1; tmp1 = -tmp1) NEXT
        put.int = get.int(single) NEXT
        IF put.int<63> => (tmp<1> = 1; put.int = -put.int) NEXT
        put.int = put.int * get.int NEXT
        DECODE get.int(single)<31> =>
                BEGIN
                put.int = put.int SRD get.int,
                put.int = put.int SLO (NOT get.int + 1)
                END NEXT
        put.int(single,0) = put.int * tmp<0> * tmp<1>
        END,

DIVFIX.ex:=             !Divide fixed point
        BEGIN{TC}
        tmp = 0 NEXT
        tmp1 = get.int(single) NEXT
        IF tmp1 EQL 0 => exception(Ill.Divisor) NEXT
        IF tmp1<31> => (tmp<0> = 1; tmp1 = -tmp1) NEXT
        tmp.d = get.int(single) NEXT
        IF tmp.d<63> => (tmp<1> = 1; tmp.d = -tmp.d) NEXT
        DECODE get.int(single)<31> =>
                BEGIN
                tmp.d = tmp.d SLO get.int,
                tmp.d = tmp.d SRD (NOT get.int + 1)
                END NEXT
        put.int(single,0) = (tmp.d / tmp1) * tmp<0> * tmp<1>
        END,

CMPS.ex:=       !Compare and Swap
        BEGIN
        tmp = get.log() NEXT
        !Serialization and Memory Lock
        tmp1 = get.log() NEXT
        tmp2 = op.addr NEXT     !Save location of second operand
        DECODE z = (tmp1 EQL get.log()) =>
                BEGIN
                (store(op.addr) = tmp1; n = tmp1 LSS{US} get.log),
                (store(tmp2) = tmp; n = 0)
                END
        !memory unlock
        END,


SIZE.ex:=
        BEGIN
        op.addr() NEXT
        put.log(0) = size(op.addr<op.size>)
        END,

SETCC.ex:=              !SET Condition Codes
        BEGIN
```

```
        eae@c@t@n@z = get.log()
        END,

REPENT.ex:=             !REPlace ENTry in map
        BEGIN
        Privilege.chk() NEXT
        tmp.d = get.int(1) NEXT !Map Entry
        tmp1 = get.log() NEXT   !Map number
        IF tmp1 GTR 1 => exception(Spec.Error) NEXT
        IF Mw[(Mptr[tmp1]<Map.addr>@'000)-4] LSS{US} get.log() =>
                exception(Spec.error) NEXT      !Check map size(from memory)
        Md[(Mptr[tmp1]<Map.addr> + get.log)@'000] = tmp.d
        !Invalidate any translation buffer associated with this entry
        !Update any copies of the map size
        END,

MAP.ex:=                !Map virtual address
        BEGIN
        Privilege.chk() NEXT
        get.log() NEXT  !address
        vp(get.log,M.n) NEXT    !translate address
        put.log(0) = vp NEXT
        put.log(0) = seg.number
        END,

WINDOW.ex:=             !window to micromachine (Implementation dependent)
        BEGIN
        !IF console.enabled => Break to microcode NEXT
        PC = PC + 1;
        Stop()
        END,

END                     !Nebula ISP description

!The instructions CMPBK, MOVTR, SCANB, ADDU, SUBU, MULU, DIVU have
!been omitted from this version of the ISP.
```

CUSTODIAN: ARMY-CR

PREPARING ACTIVITY: ARMY-CR

PROJECT: IPSC-A153

Index

ABS 73
ABSF 154
Absolute Value 73
Absolute Value Floating 154
ADD 63
Add Literal 56
Add to Message Pointer 56
Add with Carry 72
ADDC 72
ADDF 140
ADDLIT 56
ADDU 73
AMP 56
AND 77
Arithmetic Shift 79
ASH 79

BCASE 57
BCC 96
BCS 96
BEQL 93
BGEQ 95
BGTR 95
Bit Case 57
BLEQ 94
BLSS 94
BNEQ 93
BR 92
Branch 56, 92
Branch on Carry Clear 96
Branch on Carry Set 96
Branch on Equal 93
Branch on Greater than 95
Branch on Greater than or Equal 95
Branch on Less than 94
Branch on Less than or Equal 94
Branch on Not Equal 93
Branch on Truncate Clear 97
Branch on Truncate Set 97
Break 130
BRIO 56
BTC 97
BTS 97

CALL 103
Call Procedure 103
Call Supervisor 106
Call Unprivileged Procedure 104
CALLU 104
Case 57, 98
CASEIO 57
Clear 83

Clear Bit 128
Clear Floating 150
CLR 83
CLRBIT 128
CLRF 150
CMD 55
CMDL 55
CMP 87
CMPBK 120
CMPF 151
CMPS 88
CMPU 87
CMPWB 89
COB 78
Command 55
Compare (Sign Extended) 87
Compare and Swap 88
Compare Block 120
Compare Floating 151
Compare Unsigned 87
Compare within Bounds 89
Compare within Bounds and Take Exception (Range Check) 90
Convert Floating to Integer 148
Convert Integer to Floating 147
Count One Bits 78
CPSTAT 54
CSTAT 55

DBGEQ 100
DBGTR 101
DEC 71
Decrement 71
Decrement and Branch on Greater 101
Decrement and Branch on Greater than or Equal to 100
DIV 64
DIVF 144
DIVFIX 70
Divide Fixed Point 70
DIVU 75

ECODE 113
EDIV 68
EMUL 67
Enter Loop 102
ENTLP 102
EQL 91
ERET 115
ERP 116
EXCEPT 114
Exception Return 115
Exception Return and Propagate 116
EXCH 84
Exchange 84
Extended Integer Divide 68
Extended Integer Multiplication 67

FIX 148
FLOAT 147
Floating-Point Addition 140
Floating-Point Division 144
Floating-Point Multiplication 142
Floating-Point Square Root 153
Floating-Point Subtraction 141

GEQ 91
GTR 91

Halt 58

IBLEQ 99
IBLSS 100
ILIST 118
INC 71
INC2 71
INC4 71
INC8 71
Increment 71
Increment and Branch on Less than 100
Increment and Branch on Less than or Equal to 99
Indexed Load Message Pointer 55
Initiate Task 112
Insert in Doubly Linked List 118
INT 58
Integer Addition 63
Integer Division 64
Integer Modulus 66
Integer Multiplication 64
Integer Negate 65
Integer Subtraction 63
Interrupt 58
INTL 58
INVBIT 129
Invert Bit 129

JSR 106
Jump 56, 92
Jump to Subroutine 106
JUMPIO 56

LBF 126
LBFS 125
LEQ 91
LMP 55
LMPL 55
Load Bit Field (Sign Extended) 125
Load Bit Field (Zero Extended) 126
Load Message Pointer 55
Load PSW 131
Load Task 108
Logical And 77
Logical Exclusive Or 78
Logical Not 76

Logical Or 77
Logical Shift 81
Loop 99
LPSW 131
LSH 81
LSS 91
LTASK 108

MAP 134
Map Virtual Address 134
Masked Test and Branch 57
MFS 85
MOD 66
MOV 82
MOVA 83
MOVBK 121
Move Address 83
Move Arithmetic (Sign Extended) 82
Move Block 121
Move Floating 149
Move From Stack 85
Move Logical (Zero Extended) 82
Move Multiple (Fill) 121
Move To Stack 84
Move Translated 122
Move within Message 57
MOVF 149
MOVL 82
MOVM 121
MOVMSG 57
MOVTR 122
MTEST 57
MTESTL 57
MTS 84
MUL 64
MULF 142
MULFIX 69
Multiply Fixed Point 69
MULU 74

NEG 65
Negate Floating 146
NEGF 146
NEQ 91
No Operation 130
NOP 130
NOT 76

OR 77

PINIT 112
POP 86
Pop from SP Stack 86
PRAISE 111
PSTART 110
PUSH 85

Push onto SP Stack 85

RAISE 113
Raise Exception 113
RANGE 90
RDEDIT 51
RDTMSG 50
RDWCNT 50
Read 48
Read and Edit 51
Read Channel Program Status 54
Read Channel Status 55
Read to Message 50
Read with Count 50
REM 67
Remainder 67
Remainder Floating 155
REMF 155
Remove from Doubly Linked List 119
REPENT 133
Replace Entry in Map 133
RET 105
Return from Procedure 105
Return from Subroutine 107
RLIST 119
RNDI 157
ROT 80
Rotate 80
Round to Integer 157
RSR 107

SBF 124
Scan and Break 123
SCANB 123
Set Based on Condition 91
Set Bit 127
Set Condition Codes 132
Set Exception Handler Entry Address 114
SETBIT 127
SETCC 132
Size 132
SPSW 131
SQRTF 153
Start Task 110
Start Task Setting Exception 111
STASK 109
STOLIT 56
Store Bit Field 124
Store Exception Code 113
Store Literal 56
Store PSW 131
Store Task 109
SUB 63
SUBC 72
SUBF 141
Subtract with Carry 72

SUBU 74
SVC 106

TEST 90
Test Bit 129
Test Integer 90
TINIT 112
TRAISE 111
TSTART 110
TSTBIT 129

Unsigned Addition 73
Unsigned Division 75
Unsigned Multiplication 74
Unsigned Subtraction 74

WINDOW 133
Window into the Micromachine 133
WRFMSG 53
Write 52
Write from Message 53
Write Literal 53
WRLIT 53

XLMP 55
XOR 78